Contribution à la Commande des Systèmes à Événements Discrets par Filtre Logique

Romain Pichard

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

ÉCOLE DOCTORALE SCIENCES DU NUMÉRIQUE ET DE L'INGÉNIEUR

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Discipline : ***AUTOMATIQUE, SIGNAL, PRODUCTIQUE, ROBOTIQUE***

Spécialité : Automatique et Traitement du Signal

Présentée et soutenue publiquement par

Romain PICHARD

Le 30 novembre 2018

Contribution à la Commande des Systèmes à Événements Discrets par Filtre Logique

---

Thèse dirigée par **Bernard RIERA**

JURY

M. Jean-Marc ROUSSEL	Maître de conférences HDR	LURPA – École Normale Supérieure Paris-Saclay	Rapporteur
M. Laurent PIÉTRAC	Maître de conférences HDR	AMPÈRE – Institut National des Sciences Appliquées de Lyon	Rapporteur
Mme Véronique CARRÉ-MÉNÉTRIER	Professeur des Universités	CReSTIC – Université de Reims Champagne-Ardenne	Examinatrice
M. Jean-François PÉTIN	Professeur des Universités	CRAN – Université de Lorraine	Examinateur
M. Serge DEBERNARD	Professeur des Universités	LAMIH – Université Polytechnique Hauts-de-France	Examinateur
Mme Ramla SADDEM	Maître de conférences	CReSTIC – Université de Reims Champagne-Ardenne	Examinatrice Co-encadrante
M. Alexandre PHILIPPOT	Maître de conférences	CReSTIC – Université de Reims Champagne-Ardenne	Examinateur Co-encadrant
Mme Pascale MARANGÉ	Maître de conférences	CRAN – Université de Lorraine	Invitée
M. Bernard RIERA	Professeur des Universités	CReSTIC – Université de Reims Champagne-Ardenne	Examinateur Directeur de thèse

Remerciements

Les travaux de thèse présentés dans ce manuscrit ont été réalisés dans l'équipe Commande et Diagnostic des Systèmes à Evénements Discrets du CReSTIC (Centre de Recherche en STIC) de l'Université de Reims Champagne-Ardenne.

Je remercie Monsieur le Professeur Bernard RIERA, directeur du CReSTIC, de m'avoir accueilli au sein du laboratoire et d'avoir dirigé mes travaux. Etre le « berger d'un troupeau de chats » n'est pas chose aisée, mais tu as toujours su m'accorder le temps nécessaire pour répondre à mes questions. Malgré tout le sérieux que tu peux avoir, ton rire est communicatif et résonnera dans ma tête aussi longtemps que dans les couloirs du laboratoire.

Je remercie vivement les rapporteurs de ce manuscrit de thèse, Monsieur Laurent PIÉTRAC de l'Institut National des Sciences Appliquées de Lyon (laboratoire AMPÈRE) et Monsieur Jean-Marc ROUSSEL de l'École Normale Supérieure Paris-Saclay (laboratoire LURPA). Je leur suis très reconnaissant pour avoir pris le temps nécessaire à l'étude de mes travaux. Merci également pour vos remarques, aussi précises que pertinentes, qui m'ont permis d'améliorer ma présentation.

Merci à Monsieur le Professeur Jean-François PÉTIN pour avoir accepté d'examiner mes travaux deux fois, lors du comité de mi-thèse puis lors de ma thèse. Ses avis et conseils m'ont permis de clarifier la route à suivre. Merci également à Monsieur le Professeur Serge DEBERNARD pour m'avoir fait l'honneur d'examiner mes travaux et de présider ce jury de thèse. Merci à Madame Pascale MARANGÉ pour avoir initié ses travaux il y a quelques années, pour ces échanges durant toute la thèse et pour avoir participé à mon jury de thèse.

Je souhaite à présent remercier mes encadrants, en commençant par Madame Ramla SADDEM qui a accepté de suivre mes travaux, même lorsque ceux-ci s'éloignaient de sa spécialité. J'enveloppe dans ce paragraphe mes remerciements à Monsieur Alexandre PHILIPPOT, ses conseils et son aide m'ont permis d'avancer dans ce cirque parsemé de scolopendre qu'est une thèse. Le travail avec toi est chose aisée de par ta générosité et ton humanité.

Je tiens à remercier mes collègues d'équipe, Monsieur François GELLOT pour sa douce voix et son aide sur la Cellflex, Monsieur le Professeur Janan ZAYTOON pour ses mots d'encouragement et ses conseils tout au long de la thèse et lors du comité de mi-thèse, Monsieur Mohamed NIANG pour les bons moments passés ensemble à discuter science et Madame Imane TAHIRI pour son sourire et sa bonne humeur. Je remercie tout particulièrement Madame la Professeur Véronique CARRÉ-MÉNÉTRIER pour avoir dirigé cette équipe CDSED, sa bienveillance et ses mots d'encouragement quotidiens font d'elle une vraie mère pour cette équipe et ce laboratoire.

Merci à mes collègues, devenus amis, Messieurs Laurent ARCESE et Sinuhé MARTINEZ-MARTINEZ pour avoir su m'apporter leurs avis extérieurs durant cette thèse. Ces liens que nous avons créés entre deux coups de raquette, autour d'un café ou d'une pizza, ne s'effaceront pas.

Merci à Madame Ida LENCLUME pour son aide en administratif, en chocolat et en blague carambar et à Madame Sarah FILALI pour son soutien en bonbon et en discussion.

Je remercie chaleureusement ma famille pour son soutien durant ces nombreuses années d'études. Merci à vous d'avoir fait le déplacement pour cette importante journée qu'a été ma soutenance de thèse. Un merci particulier à Elvire et Edith pour m'avoir aidé à corriger les « quelques » fautes de français présentes initialement dans ce manuscrit.

Pour finir en beauté, je souhaite dédier ces travaux à la personne qui m'accompagne depuis tant d'années. Tu m'as toujours soutenu et su tirer le meilleur de moi, je ne serai pas ici sans ton aide, ton avis et ton amour. Notre nouvelle vie à trois va bientôt commencer et je suis comblé d'entamer cette aventure avec toi. Merci Léa.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Les travaux de cette thèse contribuent à une méthode formelle de conception d'un programme de contrôle/commande pour les systèmes automatisés de production (SAP) contrôlés par des automates programmables industriels (API). Les systèmes automatisés de production seront considérés comme des systèmes à événements discrets (SED) (Cassandras et Lafortune, 2009) ayant des entrées et des sorties logiques (Balemi *et al.*, 1993).

Les SAP sont, avec les opérateurs humains, les pierres angulaires de l'efficacité d'une ligne de production. Les SAP sont généralement contrôlés par des dispositifs électroniques appelés automates programmables industriels (API). La performance et la fiabilité des SAP et des API sont encore aujourd'hui des enjeux majeurs en recherche théorique et appliquée. De plus, l'arrivée de la 4^{e} révolution industrielle et des principes de l'industrie du futur imposent aux systèmes d'être toujours plus flexibles et fiables.

Les approches industrielles de conception des programmes de contrôle/commande des SAP sont très souvent basées sur la capacité de l'automaticien à anticiper et détecter efficacement les erreurs présentes dans un programme. Ces approches permettent de tendre vers un résultat acceptable après plusieurs allers-retours entre la conception et les tests. Néanmoins, elles ne peuvent garantir ni la validité ni l'optimalité du résultat.

Les approches issues du monde académique visent au contraire à garantir formellement (i.e. mathématiquement) le résultat. Ces approches formelles se basent sur une description mathématique du problème à résoudre ainsi que sur des propriétés, des théorèmes et des outils permettant la résolution automatique du problème. Néanmoins, les approches actuelles pour la conception formelle d'un programme API ne sont que peu connues et encore moins utilisées par les automaticiens de l'industrie. Ce fait peut être expliqué

par la complexité de ces approches et leur inadéquation par rapport aux pratiques de l'automaticien.

Pour répondre aux besoins de flexibilité et de fiabilité des SAP du futur, cette thèse a pour objectif de proposer une méthode et des outils formels, utilisables dans l'industrie par les automaticiens, permettant la conception d'un programme API reposant sur l'utilisation d'un filtre logique de commande.

Contributions de la thèse

L'approche proposée se base sur la notion de contrainte logique et de filtre logique pour la commande (Riera *et al.*, 2015). L'objectif des travaux est de fournir une méthode et des outils utilisables dans l'industrie permettant, à partir du cahier des charges, d'obtenir un programme API généré automatiquement et vérifié formellement. La méthode de conception proposée dans cette thèse pour l'obtention d'un filtre logique est présentée dans la figure 1.

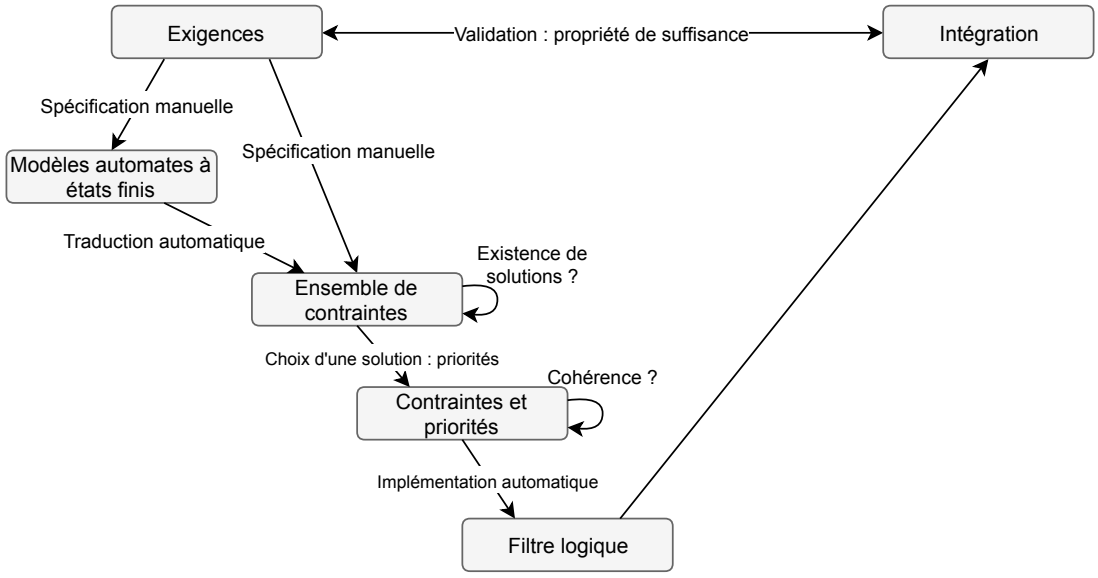


FIGURE 1 – Approche formelle proposée pour la conception d'un filtre logique

Dans le but de supporter la méthode proposée, différentes contributions théoriques et algorithmiques sont proposées et présentées succinctement ci-dessous.

— La formalisation des contraintes logiques définissant le filtre logique est retravaillée pour améliorer l'expressivité de ces contraintes.

— La notion de priorité entre les variables d'une contrainte, présentée dans des travaux précédents (Coupat, 2014), est formalisée (Pichard *et al.*, 2018b).
— La propriété de cohérence d'un filtre logique, introduite dans des travaux précédents (Riera *et al.*, 2015), est formalisée. De plus, une solution fournissant une condition nécessaire et suffisante pour la vérification formelle de cette propriété est proposée (Pichard *et al.*, 2017b).
— Dans la méthodologie de base, la définition des contraintes logiques est effectuée manuellement à partir des exigences textuelles du cahier des charges. Dans cette thèse, une approche à base d'automates à états finis est proposée pour générer automatiquement les contraintes logiques.
— L'algorithme d'implémentation « classique » du filtre logique (Coupat, 2014), est mis à jour dans cette thèse pour prendre en compte le nouveau formalisme.
— Deux algorithmes de recherche locale de solutions sont proposées en se basant sur des techniques de solveur SAT. Ces solutions d'implémentation permettent de simplifier les étapes de conception en amont de l'implémentation (Pichard *et al.*, 2018a).

Enfin, afin de valider l'approche proposée dans cette thèse et de faciliter son utilisation, différentes étapes de conception sont intégrées dans un environnement logiciel : SEDMA (Pichard *et al.*, 2017a). Ce manuscrit ne détaille pas la conception de ce logiciel. L'annexe B décrit SEDMA et les fonctionnalités disponibles.

Organisation du mémoire de thèse

Le premier chapitre permet tout d'abord d'introduire la problématique de conception d'un programme de contrôle/commande pour un automate programmable industriel (API). Puis, des approches de conception issues du monde industriel et du monde académique sont discutées. Enfin, l'approche par filtre logique retenue est présentée, des limitations sont soulevées et des contributions autour de cette approche sont proposées.

Le chapitre II a pour objectif d'illustrer et de motiver plus en détail les propos du chapitre I. Pour ce faire, deux études sont présentées et discutées. La première permet de montrer qu'il existe un manque de méthodologie pratique, lorsque l'objectif est de concevoir un

programme API. La seconde met en évidence que les méthodes formelles, issues du monde académique, ne permettent pas actuellement une conception efficace de programme API.

Le chapitre III détaille les contributions, méthodologiques et formelles, apportées à l'approche par filtre logique. Tout d'abord une méthodologie de conception, basée sur un cycle de développement en V formel, est proposée. Puis, une formalisation des éléments permettant de définir un filtre logique est présentée. Ensuite, une approche de vérification formelle, nécessaire et suffisante, de la cohérence d'un filtre logique est proposée. Enfin, des liens entre la théorie de la commande par supervision (SCT) et l'approche par filtre logique sont discutés. Cette association d'approches permet de proposer une solution de génération automatique des contraintes logiques définissant le filtre logique.

Le chapitre IV détaille les contributions liées à l'implémentation du filtre logique dans un API. Dans un premier temps, l'algorithme initial d'implémentation du filtre logique est discuté et des améliorations sont proposées. Dans un second temps, deux algorithmes de recherche locale de solutions sont proposées en se basant sur des techniques de solveur SAT.

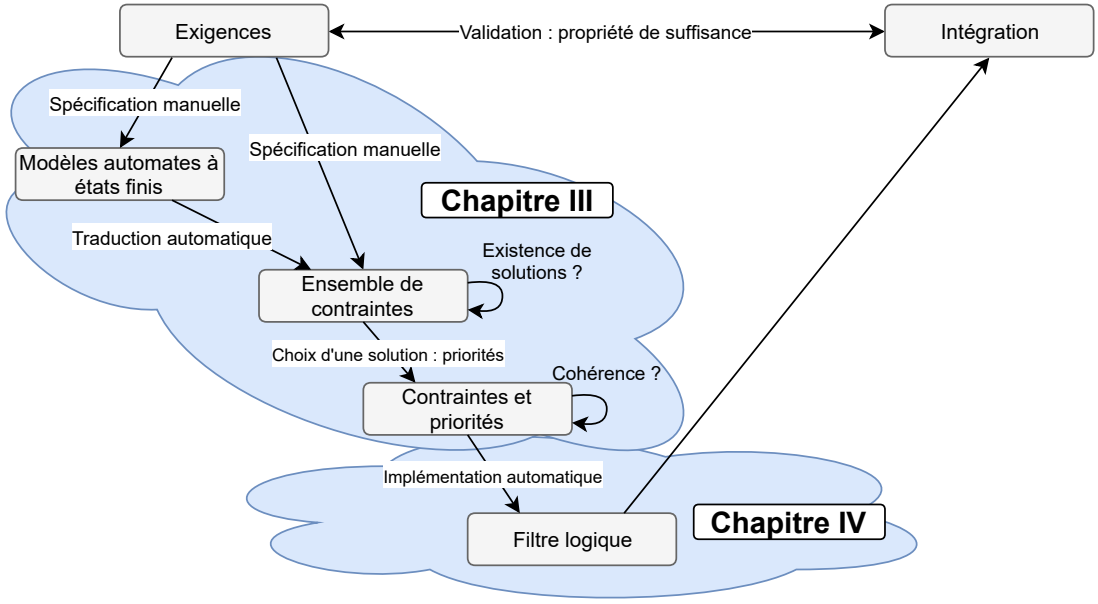


FIGURE 2 – Organisation des contributions

Enfin, le chapitre V est consacré à une application de l'approche proposée sur un système réel. Ce système comporte plusieurs sous stations commandées chacune par un API. Pour trois de ces sous stations, le cahier des charges, les contraintes logiques définissant le filtre logique, l'analyse du filtre logique et l'implémentation de celui-ci sont présentés.

CHAPITRE I

Commande des systèmes automatisés : contexte industriel et académique

I-1 La conception de contrôleur dans l'industrie du futur

Le concept « Industrie 4.0 » apparaît pour la première fois lors du salon de la technologie industrielle de Hanovre en 2011. On parle alors de quatrième révolution industrielle illustrant une nouvelle façon d'organiser les moyens de production, et ce grâce à l'émergence du digital et de la robotisation. Elle fait suite à la mécanisation dans l'industrie et l'utilisation de la machine à vapeur au 18^{e} siècle, à l'électrification et au travail à la chaîne pour la production de masse (fin 19^{e}, début du 20^{e} siècle), et à l'apparition des premiers systèmes programmés électroniquement dans les années 1970.

Pour mener à bien cette transformation, différents enjeux sont identifiés : économiques, technologiques, organisationnels, environnementaux, sociétaux [1]. Pour répondre aux enjeux technologiques, l'industrie du futur s'appuie sur différents concepts et outils :

— continuité numérique (IIoT, BigData, jumeaux numériques, etc.) ;
— cybersécurité (accès données clients, intrusion dans le système de production, etc.) ;
— flexibilité (production à la demande, moyens de production modulable, etc.) ;
— maintenance 2.0 (standardisation, opérateurs augmentés, etc.) ;

1. `http://industriedufutur.fim.net/`

— éco-usine (réduction de l'empreinte écologique, optimisation de l'énergie, etc.).

Cette thèse se focalise sur deux des enjeux technologiques liés aux systèmes automatisés de production : la flexibilité des contrôleurs et la standardisation des programmes.

I-1.1 Les besoins pour les systèmes automatisés de production du futur

Un système automatisé de production (SAP) a pour but de traiter une matière d'œuvre pour lui apporter une valeur ajoutée de façon reproductible et efficace. Il se décompose en deux parties : la partie opérative (PO) dont les actionneurs agissent sur le processus automatisé, et la partie commande (PC) qui coordonne les actions de la partie opérative.

Aujourd'hui, la majorité des installations s'appuient sur une PO non-modulable contrôlée par un programme de contrôle/commande ajustée à celle-ci. De plus, la maintenabilité et la lisibilité des programmes sont rendues difficiles par manque de méthodologie. En effet, les approches de conception de ces programmes s'appuient sur l'expertise métier des automaticiens. Enfin, ce manque de méthodologie restreint la possibilité de garantir le bon fonctionnement des contrôleurs.

Demain, dans le but de répondre aux besoins de flexibilité de l'usine du futur, les parties opératives des SAP devront gagner en modularité et les parties commandes en flexibilité et fiabilité. Les approches actuelles de conception de la PC ne permettant pas d'atteindre efficacement ces objectifs, de nouvelles propositions méthodologiques seront nécessaires. De plus, afin de garantir la fiabilité et améliorer la maintenabilité des programmes de contrôle/commande, ces approches devront être davantage formelles et standardisées.

I-1.2 Principe de la conception d'un contrôleur

Le développement d'un contrôleur, quel que soit le système à piloter, passe par plusieurs grandes étapes : l'analyse, la spécification, la conception, l'implémentation et l'intégration. Certaines de ces étapes peuvent être détaillées en sous-étapes permettant une parallélisation et un partage des tâches. Tout au long du cycle de développement sont réalisés des tests, ceux-ci permettent la vérification et la validation des étapes et du résultat final.

La vérification consiste à répondre à la question « Le contrôleur est-il bien fait ? » alors que la validation répond à la question « Le contrôleur fait-il bien son travail ? » (Boehm, 1987). Des définitions normalisées sont présentées ci-dessous.

Définition 1. ***Vérification*** *(ISO9000, 2015)*

Confirmation par des preuves tangibles que les exigences spécifiées ont été satisfaites

Définition 2. ***Validation*** *(ISO9000, 2015)*

Confirmation par des preuves tangibles que les exigences prévues pour une utilisation spécifique ou une application ont été satisfaites

Ces tests sont réalisés tout au long du développement du contrôleur. Ils permettent de s'assurer que l'étape en cours est bien réalisée mais également de confirmer que le résultat est conforme par rapport aux attentes des étapes précédentes. Foures (2015) propose une représentation générique des étapes du cycle de vie pour le développement d'un système. Nous proposons une version légèrement modifiée permettant de mettre en évidence l'étape d'implémentation obligatoire lors de la conception d'un contrôleur (Fig. 5).

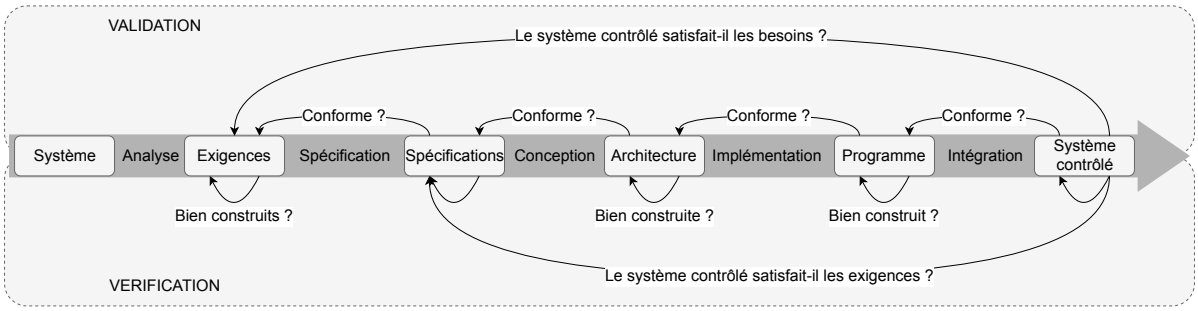


FIGURE 3 – Cycle générique de validation et vérification (Inspiré de Foures (2015))

Tout d'abord, une analyse du système doit être réalisée. Celle-ci peut être séparée en deux parties : l'analyse fonctionnelle et l'analyse dysfonctionnelle. L'analyse **fonctionnelle** permet de savoir ce que le système **peut** faire. L'analyse **dysfonctionnelle** va permettre de savoir ce que le système **ne doit pas** faire. A partir de ces analyses, les **besoins** vont pourvoir être listés, ceux-ci expriment ce que l'on **veut** faire avec le système. Ces analyses et les besoins représentent un ensemble d'exigences informelles, ces exigences sont synthétisées dans un document : le cahier des charges.

Le contrôleur doit répondre aux besoins tout en respectant les contraintes résultantes de l'analyse dysfonctionnelle. Néanmoins les exigences sont informelles et donc peuvent être comprises de différentes manières. Une étape de **spécification** des exigences est donc nécessaire, cette étape a pour objectif d'interpréter le cahier des charges, il en résulte un ensemble d'exigences formelles appelées spécifications. Ces spécifications, à l'opposé du cahier des charges, ne peuvent être interprétées que d'une seule manière.

A partir des spécifications, une conception architecturale est réalisée permettant la mise en relation des fonctions principales du contrôleur. Puis, chaque fonction est détaillée en sous-fonctions afin de faciliter leur conception.

En fonction de l'architecture matérielle liée au système et des besoins, un langage informatique cible est déterminé. Chaque fonction peut alors être traduite vers ce langage cible, cette phase s'appelle **l'implémentation** du contrôleur.

Une fois le contrôleur vérifié et validé d'un point de vue logiciel, il doit être intégré dans l'environnement matériel du système. Cette étape d'**intégration** consiste à relier le contrôleur physique, contenant le programme, au système à contrôler.

Différentes méthodologies existent permettant l'organisation de ces étapes. Dans le monde industriel, les méthodes utilisées sont encore souvent basées sur l'expérience et les compétences techniques des acteurs du développement. Ces méthodes permettent une étape de conception relativement rapide mais demandent plus d'effort au niveau de l'implémentation et des tests. Les méthodes **formelles**, principalement utilisées dans le monde académique, cherchent à décrire mathématiquement le problème à résoudre. Ces méthodes permettent une automatisation de certaines des étapes précédemment décrites pour une meilleure fiabilité du résultat. Néanmoins, les méthodes formelles demandent en général un niveau de connaissance théorique élevé et une durée allongée pendant l'étape de conception (Frey et Litz, 2000).

Dans la section suivante, des méthodologies et outils utilisés dans l'industrie pour la conception d'un contrôleur sont présentés.

I-2 La conception d'un contrôleur : contexte industriel

Dans un contexte industriel, les méthodologies principalement utilisées pour le développement d'un contrôleur sont souvent tirées, d'une part de l'ingénierie système et d'autre part des méthodes de gestion de projet logiciel. Ces méthodologies sont orchestrées de différentes manières selon l'organisation des différentes étapes (analyse, spécification, conception, implémentation et intégration) et des tests. Dans la suite de cette section sont présentées successivement, les principes de l'ingénierie système (section I-2.1), des méthodes cycliques (section I-2.2.1) et des outils d'aide à la conception (section I-2.3). Enfin, les apports des méthodes *formelles* dans les cycles de développement sont présentés (section I-2.4).

I-2.1 Normes pour l'ingénierie système et les API

L'ingénierie système (IS) peut être définie comme « une démarche méthodologique pour maîtriser la conception des systèmes et produits complexes » [2]. Les règles orchestrant cette démarche d'IS sont détaillées dans des normes. Celles-ci sont ensuite réalisées à l'aide de méthodes, supportées par des outils (Fig. 4). En France, l'Association Française d'Ingénierie Système (AFIS) a pour vocation d'encadrer et de promouvoir l'application de l'IS.

La **normalisation** ou la **standardisation** est le fait d'établir respectivement des normes et standards techniques. Ce sont des référentiels communs et documentés destinés à harmoniser l'activité d'un secteur. Ceci est réalisé par des organismes spécialisés, qui sont le plus souvent des organismes nationaux (AFNOR [3], ANSI [4]), ou internationaux (ISO [5], IEC [6]).

Les principales normes de l'IS, d'après l'AFIS [7] sont présentées ci-après.

2. http://www.afis.fr/nm-is/default.aspx
3. https://www.afnor.org/
4. https://www.ansi.org/
5. https://www.iso.org/fr/home.html
6. http://www.iec.ch/
7. http://www.afis.fr/nm-is/Pages/Normes%20IS/Normes%20d%e2%80%99Ing%c3%a9nierie%

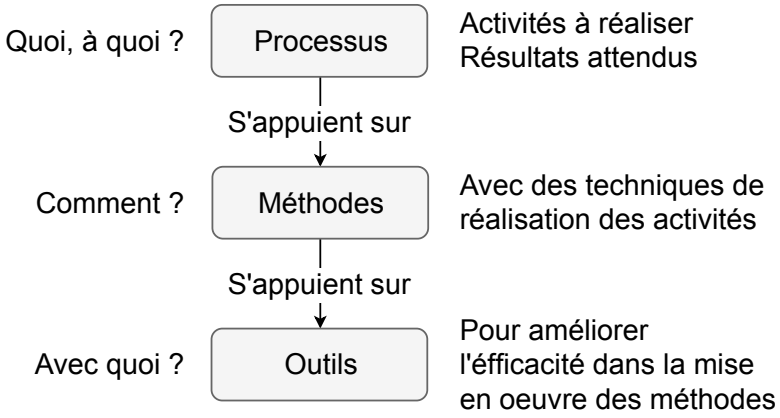


FIGURE 4 – Principe de l'IS (inspiré de l'AFIS)

— IEEE 1220 : Cette norme se focalise sur la conception du système (de l'analyse des exigences jusqu'à la définition physique du système).
— EIA 632 : Cette norme étend les principes de l'IEEE 1220 en intégrant le transfert du système réalisé vers son utilisateur final.
— ISO 15288 : Cette dernière intègre l'ensemble du cycle de vie d'un système (produit), de sa conception à son retrait. Celle-ci apparait, aujourd'hui, comme la référence.

Le contrôle/commande d'un système est une sous-partie du processus global de l'IS, certaines normes spécifiques existent alors en fonction des domaines d'applications. En ce qui concerne les Automates Programmables Industriels (API), servant à contrôler les systèmes automatisés de production, la principale norme est l'IEC61131 (2018). Celle-ci définit les différents aspects liés à un API : exigences matérielles, langages de programmation, communications... Le principe de fonctionnement et les langages de programmation propres aux API seront présentés plus loin dans ce chapitre (section I-3.1.2).

Bien que ces normes encadrent l'utilisation, par exemple en limitant les langages de programmation (IEC61131-3, 2013), elles n'imposent pas la façon de les utiliser. Concrètement, deux constructeurs d'API différents, peuvent utiliser des structures de données et des fichiers de sauvegarde de natures différentes pour un même langage de programmation. De même, deux développeurs de programmes peuvent réaliser une même fonction de différentes manières.

Dans le but de faciliter l'interopérabilité, la flexibilité, la maintenance ou bien encore la

20Syst%c3%a8me.aspx

mise à jour d'un système, il est possible d'établir des standards afin d'uniformiser l'utilisation de ces normes. Un standard est un ensemble de recommandations ou de préférences, en général proposé par un groupe d'utilisateurs spécialisés du domaine.

En ce qui concerne la programmation des API, un standard de référence est PLCopen [8]. Celui-ci se base principalement sur la norme IEC61131-3 (2013) décrivant les langages de programmation pour les API. PLCopen propose différentes règles de conception et syntaxiques, décrivant des « bonnes pratiques » à suivre pour la conception d'un programme API. Le respect de ces règles permet de limiter le risque d'erreurs dans un programme, ainsi que de simplifier la lisibilité et la maintenabilité d'un programme.

I-2.2 Méthodes de conception

En fonction de l'organisation des étapes de développement et des tests, différentes méthodes ont été proposées. Cette section présente deux catégories de méthode : les méthodes cycliques et les approches agiles.

I-2.2.1 Méthodes cycliques

Les méthodes cycliques sont les plus connues pour la gestion de projet et la conception de système. Elles diffèrent principalement en fonction de leur structuration et de leur souplesse (vs. rigidité). La structuration correspond à l'organisation hiérarchique des étapes de développement et de la place des tests dans cette organisation. La souplesse est liée à la possibilité de réagir efficacement à la détection d'une erreur, ou bien de la modification des exigences du client.

Historiquement, le cycle en cascade a été proposé pour la gestion de projet dans le domaine du bâtiment (Royce, 1987). Ce cycle en cascade est basé sur deux principes :

— une phase est débutée uniquement si la précédente est achevée ;
— la modification d'une phase a un fort impact sur les phases suivantes.

Cette méthodologie est très linéaire, donc simple à comprendre et à mettre en place. De plus, elle permet en théorie, une estimation précise de la date de fin de projet. Néanmoins,

8. `http://www.plcopen.org/index.html`

elle comporte de nombreux inconvénients comme en particulier la très faible tolérance aux erreurs. En effet, les anomalies sont détectées tardivement et impliquent obligatoirement un retour à une phase précédente voire, au début du cycle. L'exemple du bâtiment est édifiant : une erreur détectée sur les fondations après la construction du toit implique en général la reprise complète du projet.

Afin de résoudre le problème de rigidité du cycle en cascade, le modèle en V [9] a été proposé (Fig. 5). Avec ce modèle, chaque phase descendante est mise en relation avec les tests, ceci permet de détecter plus rapidement les erreurs et de les corriger avant de passer aux phases suivantes.

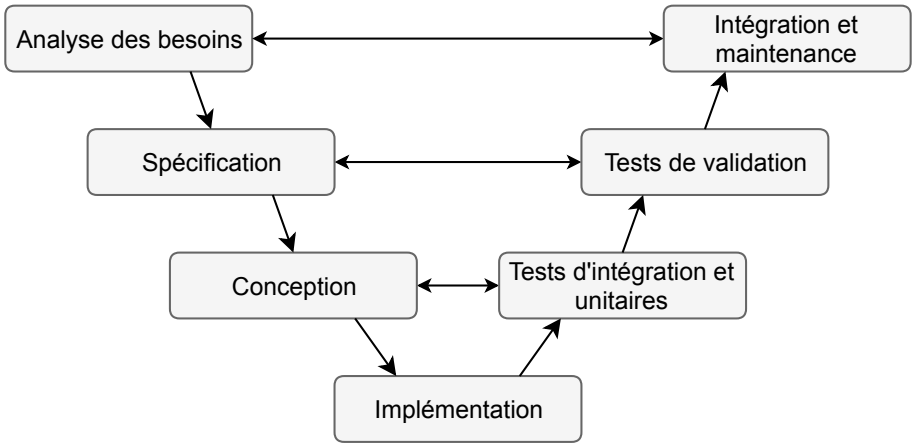


FIGURE 5 – Cycle en V

D'un point de vue théorique, cette organisation permet de tendre vers un produit entièrement conforme aux attentes du client. Néanmoins, dans un souci de gain de temps et donc de réduction des coûts de développement, certaines phases sont réduites voire supprimées.

La phase de spécification est souvent la première à être mise de côté. Comme dit précédemment, cette phase a pour but de formaliser les exigences et envies du client afin d'obtenir des spécifications, néanmoins elle demande beaucoup de temps (aller/retour entre le client et l'équipe de spécification) ainsi qu'une maîtrise des outils formels permettant la représentation de tous types d'exigences.

Les attendus des tests sont normalement définis pendant les phases descendantes, cela permet d'oublier le minimum d'éléments car les tests à effectuer sont rédigés en même temps que la conception de la fonction. Il s'avère que d'un point de vue pratique cela

9. http://www.geek-directeur-technique.com/2009/02/04/le-cycle-en-v

rallonge encore la phase de conception. Dans le but de réduire ce temps de conception (descente du cycle), les tests ne sont généralement rédigés et effectués qu'après l'étape d'implémentation.

Dans la pratique, la suppression de l'étape de spécification permet d'aboutir plus rapidement à un résultat (après l'étape d'implémentation). Néanmoins cela reporte les problèmes à d'autres niveaux. En effet, le manque de formalisation doit se retrouver compensé par la capacité des chefs de projets à anticiper les problèmes et à bien comprendre les attentes des clients. De plus, le fait de ne pas avoir de tests organisés et hiérarchisés impose aux équipes de développement d'être très rigoureuses et performantes. Dans le cas contraire, le temps gagné dans la phase de conception risque d'être perdu à cause des nombreuses itérations de tests/codages.

Bien que le cycle en V permette de mieux réagir aux erreurs que le cycle en cascade, celui-ci ne permet pas d'intégrer efficacement le client dans les étapes de conception. Par conséquent, un changement des exigences par le client risque d'allonger fortement le temps global du projet. Pour répondre à ce problème le cycle en spirale a été proposé (Boehm, 1988).

Le cycle en spirale reprend les phases du cycle en V, mais prévoit l'implémentation de versions successives, ce qui permet de mettre l'accent sur la gestion des risques, la première phase de chaque itération étant dédiée à ce poste. Le cycle en spirale prévoit la livraison de prototypes (versions intermédiaires) pouvant s'agir de programmes partiellement fonctionnels : à chaque fonction principale peut correspondre un prototype.

Ce cycle apporte une grande flexibilité. En effet, si chaque prototype apporte des fonctionnalités indépendantes, il est possible de changer l'ordre de développement des versions. De plus, le risque final est réduit de par la présence accrue du client pendant le processus. En effet, le client peut tester chaque prototype et ainsi n'a pas besoin d'attendre la version finale pour faire remonter un problème.

Néanmoins chaque cycle comporte de nombreuses phases et peut être compliqué à organiser. En réduisant le nombre de phases d'un cycle, il est possible de définir le cycle itératif (Fig. 6).

Tout commence par l'expression des besoins : le client explique ce qu'il veut, tout en

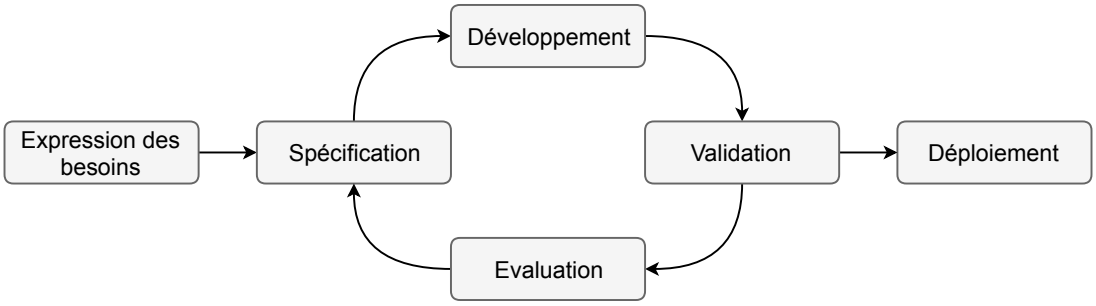


FIGURE 6 – Cycle itératif

sachant que ces besoins peuvent être modifiés par la suite du processus. Ensuite, le processus itératif en lui-même commence avec la rédaction des spécifications qui est suivie par le développement (implémentation), puis la validation (tests) et enfin une évaluation du travail qui servira d'information de départ pour le cycle suivant (difficultés rencontrées, fonctionnalités abandonnées, ...). Le déploiement est réalisé à l'issue de la validation : les livrables (il peut s'agir d'une version, ou d'une documentation) qui ont été validés sont déployés, c'est-à-dire mis à disposition.

Le cycle itératif est le plus souple de tous ceux présentés jusqu'ici. Chaque itération permet de s'adapter à ce qui a été appris dans les itérations précédentes et le projet fini peut varier du besoin qui a été exprimé à l'origine. Comme dans le cycle en spirale, la mise à disposition de livrables à chaque cycle permet un apprentissage de l'utilisateur final en douceur.

Basé sur le même besoin d'association du client dans le cycle de développement du système, les approches agiles ont été proposées. La section suivante présente les principes fondamentaux de ces approches agiles et quelques méthodes les mettant en œuvres.

I-2.2.2 Approches agiles

Les approches agiles [10] sont basées sur des valeurs et des principes, permettant de renforcer au maximum les liens entre clients et développeurs. Une description générale de ces approches a été rendue publique en 2001 (`http://agilemanifesto.org/`). Les valeurs fondamentales d'une approche agile sont les suivantes :

— **les individus et leurs interactions** plus que les processus et les outils ;

10. `https://www.agiliste.fr/introduction-methodes-agiles/`

— **des logiciels opérationnels** plus qu'une documentation exhaustive ;
— **la collaboration avec les clients** plus que la négociation contractuelle ;
— **l'adaptation au changement** plus que le suivi d'un plan.

En plus de ces valeurs, 12 principes généraux sont énoncés :

— satisfaire le client en priorité ;
— accueillir favorablement les demandes de changement ;
— livrer le plus souvent possible des versions opérationnelles de l'application ;
— assurer une coopération permanente entre le client et l'équipe projet ;
— construire des projets autour d'individus motivés ;
— privilégier la conversation en face à face ;
— mesurer l'avancement du projet en termes de fonctionnalités de l'application ;
— faire avancer le projet à un rythme soutenable et constant ;
— porter une attention continue à l'excellence technique et à la conception ;
— faire simple ;
— responsabiliser les équipes ;
— ajuster à intervalles réguliers son comportement et ses processus pour être plus efficace.

Basées sur ces valeurs et principes, différentes méthodes agiles ont été proposées : Scrum (Schwaber, 2004), eXtreme Programming (XP) (Beck et Gamma, 2000), Adaptive Software Development (ASD) (Highsmith, 2013)... Différentes méthodes existent car il n'y a jamais une seule façon d'interpréter des principes et de mettre en place des valeurs.

Scrum et XP sont les plus connues, et peuvent être utilisées ensemble car elles sont complémentaires [11] (Schwaber et Mar, 2002). En effet, Scrum se positionne au niveau de la gestion et de l'organisation de projet là où XP se positionne au niveau des activités de développement.

Pour résumer ces méthodes, Scrum adresse la problématique du processus de production du logiciel, l'organisation et la dynamique du projet. Le développement incrémental, néanmoins, suppose des pratiques d'ingénierie logicielle rigoureuses et efficaces, que Scrum ne couvre pas.

11. https://blog.xebia.fr/2008/01/10/scrum-ou-xp-scrum-et-xp/

XP quand à elle, se focalise sur le développement. Dans XP, le test automatisé n'est pas seulement vu comme une bonne pratique du développement agile : il en est l'épine dorsale. Cette méthode a donné naissance au principe du TDD (Test Driven Development) : le développeur écrit en premier lieu un test automatisé de la fonctionnalité qu'il souhaite implémenter, développe ensuite juste assez de code pour satisfaire le test, et finalement remanie le code pour améliorer le design et supprimer la duplication.

Afin d'aider le concepteur de programme, différents outils intègrent les normes, standards et méthodes présentés précédemment. La section suivante présente quelques outils liés à la conception d'un programme API.

I-2.3 Outils d'aide à la conception

Cette section présente trois concepts permettant d'aider l'automaticien à concevoir un programme API. Pour chacun de ces concepts, quelques outils les intégrant sont proposés.

I-2.3.1 Mise en service virtuelle (virtual commissioning)

Quelles que soient les méthodes utilisées, des tests sur le système sont nécessaires. D'un point de vue gestion de projet, l'avancement du développement du programme API peut donc être ralenti voire arrêté si le système réel n'est pas encore construit ou disponible pour des tests. Afin de palier à ce problème, et dans le but de gagner du temps, il est possible d'utiliser des outils et techniques de *virtual commissioning* (mise en service virtuelle).

Le principe du *virtual commissioning* est d'utiliser un simulateur du système physique, et de lui associer le programme API réel. Dans ces conditions, l'API réel peut être testé et validé sans avoir besoin du système réel. Cette technique présente plusieurs avantages :

- gain de temps : possibilité d'accélérer le temps dans la simulation, les tests sont effectués avant/pendant la construction du système réel ;
- gain d'exhaustivité : il est possible d'automatiser tout ou partie des tests ;
- diminution des risques lors des tests : il est possible de tester des situations dangereuses, qu'il ne serait pas possible de tester sur le système réel.

La principale difficulté d'utilisation d'une telle technique est la modélisation du système

réel (Lee et Park, 2014), celle-ci peut prendre du temps afin d'être complète. Néanmoins, les fabricants de machines utilisent de plus en plus des modélisations 3D, et multi-physiques, pour la conception de leurs machines (Catia, SolidWorks, MapleSoft...). Les logiciels permettant d'effectuer du virtual commissioning (DELMIA, SIMAC, NX/MCD), proposent en général l'import automatique de ces modèles de conception 3D. Cela diminue donc fortement le temps de modélisation, nécessaire au virtual commissioning. Enfin, de plus en plus de fournisseurs proposent les modèles 3D de leurs éléments de parties opératives (par exemple FESTO).

Pour le moment, ces outils sont principalement utilisés pour les systèmes « critiques », par exemple dans le ferroviaire (Niang *et al.*, 2017; Coupat *et al.*, 2018). Néanmoins, dans le contexte de l'industrie du futur où les systèmes et les programmes doivent être toujours plus robustes et flexibles, cette approche, à travers le concept de « jumeau numérique », apparaît comme étant de plus en plus nécessaire (Pellicciari *et al.*, 2009).

I-2.3.2 Qualimétrie de code automate

Dans l'industrie du futur, la continuité numérique est un élément très important. Pour un programme API, cette continuité numérique nécessite avant tout un programme compréhensible et lisible. Dans le but de faciliter la lisibilité, et donc la maintenabilité, d'un programme API, il est nécessaire de suivre différentes règles de programmation et d'écriture. Ces règles peuvent concerner, la syntaxe pure (commentaires, nommages des variables...), l'architecture (découpage du programme en POU (Program Organization Unit) et en sous-fonctions, langages utilisés...), ou bien encore la complexité. Ces règles peuvent être nombreuses et difficiles à évaluer (pour la complexité par exemple).

Le standard PLCopen, présenté précédemment (section I-2.1), recense de nombreuses règles de ce genre. Basé sur ce standard, le logiciel PLC checker [12] permet d'automatiser la vérification de ces règles pour un programme API. L'outil prend en entrée, une archive complète du programme API, et fournit en sortie, une liste d'avertissements et d'erreurs (en fonction de la criticité des règles) permettant au concepteur d'améliorer son programme. Il est également possible de créer ses propres règles, permettant d'ajuster la

12. `http://www.itris-automation.fr/plc-checker/`

vérification aux exigences de l'utilisateur final.

I-2.3.3 Génération automatique de code API

Dans le but de simplifier l'application des règles d'écriture liées aux normes et standards, mais également de réduire le risque d'erreurs liées à l'étape d'implémentation, il existe des solutions permettant la génération de tout ou partie du programme API et des documentations associées. Différentes techniques de génération automatique de codes et de documentations existent (Coupat, 2014) et différents outils logiciels permettent leur application (Bajovs *et al.*, 2013).

La solution Odil de la société Prosyst [13] est un exemple de logiciel pour la génération automatique de codes et de documentations. Les industriels sont de plus en plus attirés par ce type de solution : PSA, Michelin, Bombardier, Schneider Electric, SNCF...

La société Iris Automation [14] propose une autre solution pour l'aide à la normalisation des programmes API : PLC Converter. Cette solution permet la conversion automatique d'un programme API provenant d'un environnement automate vers un autre environnement automate. La gestion des versions et des équivalences de syntaxes entre les différentes marques d'automates est effectuée automatiquement. Cette société propose également la solution PLC DocGen pour la génération automatique des documentations.

La normalisation des fonctionnements, la standardisation des pratiques, et la simplification des tests à l'aide d'outils, sont des points cruciaux pour la vérification et la validation d'un programme API. Néanmoins, les tests ne sont jamais exhaustifs, les standards sont basés sur des règles informelles, et les normes encadrent l'utilisation mais ne garantissent pas les fonctionnements. Dans le but de pouvoir garantir à 100% le bon fonctionnement d'un programme API, des méthodes formelles doivent être utilisées durant la conception et les tests. La section suivante (section I-2.4), présente le principe des approches formelles et leurs utilisations dans la conception d'un programme API.

13. `http://www.prosyst.fr/index.html`
14. `http://www.itris-automation.fr/`

I-2.4 Les méthodes formelles dans le cycle de vie

Les méthodes de conception présentées dans la section I-2.2, sont toutes basées sur un principe de vérification et de validation d'un programme développé « à la main » par raffinements successifs. Le poids de la réussite d'un projet de conception de contrôleur repose donc en grande partie sur l'ingénieur en automatisme : interprétation des spécifications informelles et codage du contrôleur. La conception du contrôleur est en général assistée par des outils standardisés de programmation. Néanmoins, les exigences informelles doivent être traduites manuellement et intuitivement dans le programme de commande étant donné que l'étape de spécification n'est pas effectuée par souci de temps et d'expertise.

Ces approches entraînent la plupart du temps des coûts de développement supplémentaires dus aux erreurs d'interprétations des spécifications informelles (Johnson, 2007). Afin de réduire au maximum le risque d'erreurs lors de la conception d'un contrôleur, il est nécessaire d'utiliser des méthodes et outils formels permettant de garantir le bon fonctionnement d'un programme API.

I-2.4.1 Principe

Une méthode peut être qualifiée de formelle lorsque, d'une part la notation est formelle, c'est-à-dire définie mathématiquement et donc sans ambiguïté. D'autre part les tests doivent être formels, c'est-à-dire que le traitement des modèles exprimés dans la notation formelle est automatisé.

Lors d'un développement classique (comme présenté précédemment), les erreurs sont en général détectées tardivement. Utiliser une approche formelle va permettre de détecter des ambiguïtés et des erreurs au plus tôt lors des différentes phases. Les phases amont sont par conséquent plus coûteuses, mais cela permet un développement global plus efficace. La figure 7 étudie la corrélation entre l'effort alloué aux phases amonts et le respect du budget global du projet (Bowen et Hinchey, 2005). Chaque point correspond à un projet terminé de la NASA, cette étude permet de mettre en évidence que plus les phases d'analyse et de formalisation sont importantes, plus le budget est respecté.

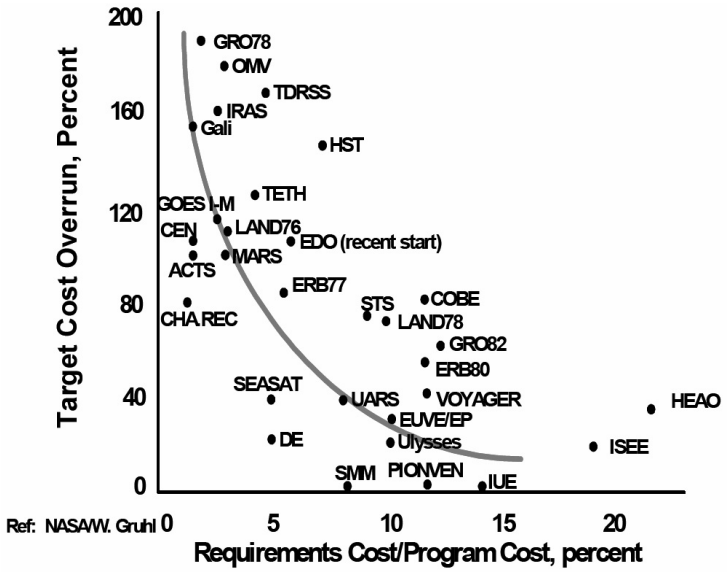


FIGURE 7 – Projets NASA : corrélation effort phases amont / respect du budget

I-2.4.2 Différents niveaux d'applications

En fonction des domaines et des besoins, les méthodes formelles peuvent être appliquées à différents niveaux lors du cycle de développement. Cette section présente la version formelle du cycle en V (Fig. 8).

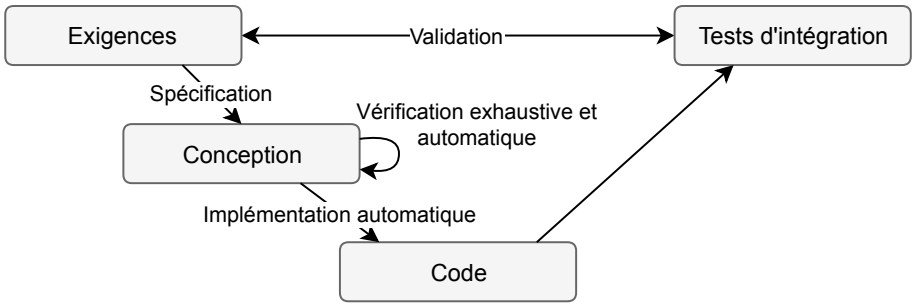


FIGURE 8 – Cycle en V formel

Spécification La spécification est une étape très importante et difficile quelles que soient les méthodes formelles utilisées (Piétrac, 2018). Celle-ci doit déboucher sur une représentation mathématique de l'ensemble des exigences informelles. C'est une étape généralement effectuée manuellement, elle demande un niveau d'expertise élevé et des compétences théoriques fortes afin d'être capable de transformer correctement n'importe quelle exigence informelle en spécification formelle.

Conception Dans cette version du cycle en V la conception est une étape purement formelle. L'entrée de cette étape est un ensemble de modèles mathématiques traduisant les exigences informelles (spécifications). L'objectif est de réussir à obtenir un modèle mathématique du résultat. Cette transformation est appelée **synthèse** et peut être considérée comme une sous-étape de la conception.

Vérification La vérification doit permettre de garantir que le modèle est bien construit, tant d'un point de vue structurel que syntaxique. Avec les approches classiques, la vérification est le plus souvent réalisée par des tests et de la simulation (Foures, 2015). La vérification est par conséquent rarement exhaustive et impose une réflexion sur chaque nouveau projet pour produire les tests nécessaires.

Selon le formalisme utilisé, il existe différentes propriétés mathématiques permettant de tester automatiquement les modèles. Ces propriétés sont utilisées afin de vérifier les modèles au plus tôt et sans effort manuel. Les approches formelles vont donc permettre une vérification exhaustive et automatisée pendant la conception. Les tests classiques, normalement effectués sur une version du produit pendant les phases « montantes », sont donc supprimables lors de l'utilisation d'approches formelles pendant la conception.

Implémentation automatique de code A partir du modèle formel issu de la phase de conception, il est possible d'implémenter (générer) automatiquement le code correspondant. Cela permet d'éviter les erreurs liées au codage manuel et donc de gagner en productivité. De plus, le générateur de code peut être lui même certifié conforme voire normé par rapport à un domaine d'application spécifique.

Validation La validation doit permettre de garantir que le contrôleur répond effectivement aux attendus. Étant donné que le contrôleur est généré automatiquement à partir de son modèle, il est possible de valider le modèle plutôt que le contrôleur. Il existe différentes méthodes, permettant de tester des propriétés de façon exhaustive sur les modèles, afin de les valider (Foures, 2015). À la différence des propriétés mathématiques utilisées lors de la vérification, les propriétés sont en général exprimées dans un langage logique lors de la validation. Ces propriétés pourront être testées exhaustivement mais généralement au

prix d'un temps de calcul assez long.

I-2.4.3 Compromis effort/exhaustivité

L'utilisation d'une méthode formelle implique un effort (temporel, d'expertise, financier) plus ou moins important, en fonction du niveau de fiabilité souhaité (Fig. 9). Néanmoins comme dit précédemment (Fig. 7), le temps « perdu » lors de la spécification est en général très largement regagné dans les phases suivantes (vérification, génération de code).

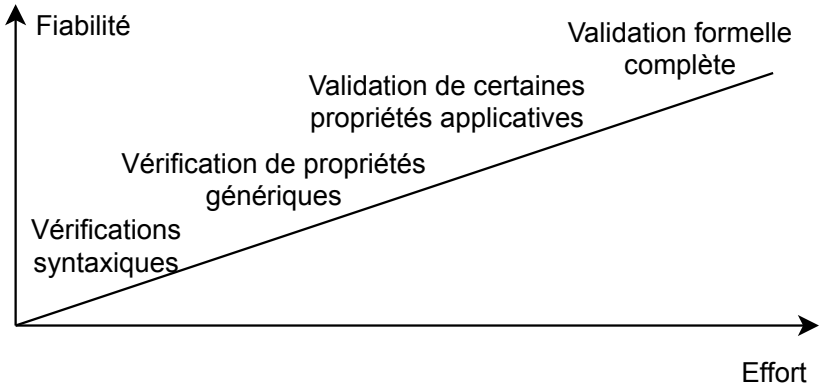


FIGURE 9 – Compromis effort/fiabilité

En fonction du domaine d'application et des normes en vigueur, l'application des méthodes formelles est plus ou moins obligatoire. Par exemple, dans le domaine ferroviaire l'utilisation des méthodes formelles est recommandée et dans le domaine aéronautique ou militaire cela est quasi-obligatoire afin de garantir la sécurité. Afin de classifier les applications d'un point vue sécurité, le standard EAL (Evaluation Assurance Level) est souvent utilisé. Celui-ci est organisé selon des *critères communs* (ISO15408, 2009) :

- EAL1 : testé fonctionnellement
- EAL2 : testé structurellement
- EAL3 : testé et vérifié méthodiquement
- EAL4 : conçu, testé et vérifié méthodiquement
- EAL5 : conçu de façon semi-formelle et testé
- EAL6 : conception vérifiée de façon semi-formelle et système testé
- EAL7 : conception vérifiée de façon formelle et système testé

Il apparaît avec ce standard que les méthodes formelles permettent de garantir un haut

niveau de fiabilité et de sécurité.

I-2.5 Bilan

Dans le monde industriel, les approches permettant d'obtenir rapidement un résultat sont privilégiées. Celles-ci sont principalement basées sur l'amélioration des versions successives du contrôleur, en se basant sur des tests à différents niveaux de l'avancement du projet. Différents outils logiciels permettent de faciliter ces tests. Néanmoins, les tests sont rarement exhaustifs, par conséquent garantir l'exactitude et la complétude du résultat est impossible.

L'introduction des méthodes formelles, dans les approches cycliques, permet d'augmenter la fiabilité du résultat. Cependant, les méthodes formelles se basent sur des définitions et des théories mathématiques. La mise en place de ces théories nécessite donc du temps de recherche fondamentale, avant de pouvoir être utilisées dans un projet industriel. Un des enjeux du monde académique, est donc de proposer des théories et outils formels permettant la modélisation des problématiques rencontrées dans le monde industriel.

Nous proposons dans la suite de ce chapitre, de présenter différentes approches académiques formelles, fournissant des solutions pour les problèmes liés à la commande des systèmes automatisés de production.

I-3 Commande des systèmes à événements discrets

L'automatique est la science des systèmes dynamiques. Pour un automaticien, un système est un ensemble d'éléments en interaction mutuelle et en interaction avec l'environnement, organisés en fonction d'un même but pour parvenir à une même fin. Le but de l'automatique est de fournir des outils de modélisation et d'analyse formels permettant de comprendre et de commander des systèmes réels tels que : réseaux de transports, atelier de production, réseaux de communications, procédés chimiques, régulation, etc. Ces différents systèmes peuvent avoir des caractéristiques très différentes et donc demander des outils de modélisation différents. Il existe plusieurs catégories de systèmes parmi lesquels nous pouvons citer :

— systèmes à temps continu : systèmes dont l'état change en permanence en fonction du temps ;
— systèmes à temps discret : systèmes dont l'état change en fonction du temps mais de manière discontinue ;
— systèmes à événements discrets : systèmes dont l'état change suite à l'occurrence d'événements ;
— systèmes hybrides : systèmes dont la modélisation nécessite l'utilisation des techniques liées aux systèmes continus et aux systèmes à événements discrets.

Dans le cadre de cette thèse, nous souhaitons proposer des méthodes et outils pour la commande des systèmes de production dans l'industrie du futur. Bien que les systèmes de production soient composés d'éléments évoluant de manière continue dans le temps, l'évolution croissante de la complexité de ces systèmes impose la prise en compte de phénomènes événementiels. Ces phénomènes sont, par exemple, la synchronisation de tâches ou d'informations, l'exclusion mutuelle d'actions ou bien encore la séquentialité d'événements. Ces phénomènes sont par nature discrets dans le temps, les approches événementielles ou hybrides permettent donc plus facilement la modélisation de tels systèmes de production.

Le cadre théorique choisi est celui des Systèmes à Événements Discrets (SED), la suite de cette section introduit les SED et les méthodes qui leurs sont liées.

I-3.1 Les systèmes à événements discrets

I-3.1.1 Notions d'événements et vocabulaire

Considérons un système ayant en entrée et en sortie plusieurs signaux logiques (Fig. 10), ce système est appelé Système à Événements Discrets (SED) si l'évolution de son état interne dépend des changements d'états des signaux d'entrées/sorties (Cassandras et Lafortune, 2009). Les changements d'états des signaux (fronts montants et fronts descendants) sont appelés « événements » et sont considérés comme discrets (instantanés).

Un événement peut être *contrôlable* (Ramadge et Wonham, 1989), *commandable* (Brandin et Wonham, 1994; Zaytoon, 2005) ou bien encore *observable* (Sampath *et al.*, 1996). Les notions d'observabilité et de diagnostic ne seront pas discutées dans ce travail de thèse.

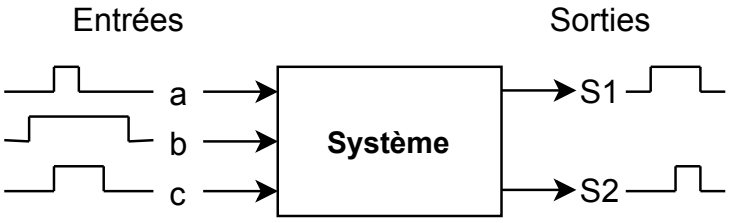


FIGURE 10 – Système à événement discret

Par contre, la différence entre contrôlable et commandable sera discutée dans le chapitre III.

Avant de détailler différentes approches permettant l'obtention d'un programme de contrôleur, le principe de fonctionnement des contrôleurs physiques principalement utilisés dans l'industrie est présenté : les automates programmables industriels.

I-3.1.2 Automates programmables industriels

Les Systèmes Automatisés de Production (SAP), ainsi que leurs contrôleurs, peuvent être modélisés par des SED ayant des entrées logiques et des sorties logiques (Balemi *et al.*, 1993). Considérons maintenant un SAP et son contrôleur (Fig. 11). Du point de vue du contrôleur, les capteurs du système sont ses entrées et les actionneurs du système sont ses sorties. Le but d'un contrôleur est alors de réagir à des changements d'états des capteurs afin d'envoyer des ordres aux actionneurs du système pour réaliser l'objectif de production.

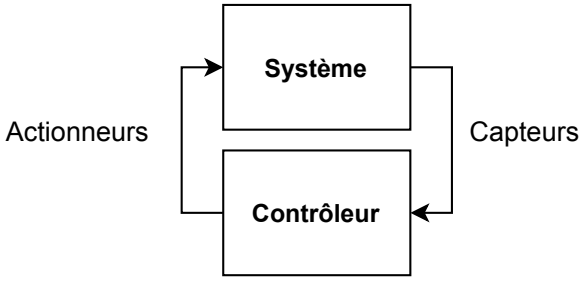


FIGURE 11 – Système à événement discret sous contrôle

Les contrôleurs principalement utilisés dans l'industrie sont des Automates Programmables Industriels (API). Après une étape d'initialisation, un API réalise trois tâches de façon séquentielle et cyclique : lecture des entrées, exécution du programme et mise à jour des sorties (Fig. 12). Le fonctionnement des API, en termes de communication, programmation et équipements, est normalisé (IEC61131, 2018).

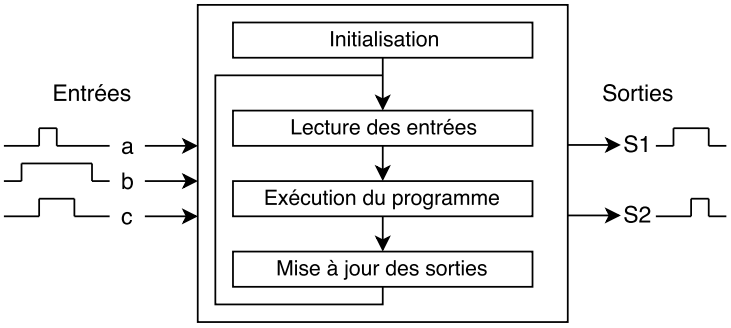


FIGURE 12 – Principe de fonctionnement d'un API

La période d'exécution d'un API (appelé cycle automate) peut être constante (balayage périodique) ou bien variable (balayage cyclique). Dans le premier cas, le cycle est redémarré au bout d'un certain temps constant (la période), si une autre tâche était déjà en cours d'exécution elle est annulée. Dans le second cas, le cycle est redémarré après avoir terminé l'étape de mise à jour des sorties, dans ce cas chaque cycle peut avoir un temps d'exécution différent. Dans les deux cas, pour que l'automate puisse contrôler de manière réactive le système, son temps de cycle doit être petit par rapport à la constante du système sous contrôle. Les temps de cycles classiques, pour les systèmes manufacturiers, sont de l'ordre de 10ms.

Le programme automate permet le calcul « temps réel » des nouvelles valeurs des signaux de sorties, ces dernières dépendent de l'état actuel de l'API (variables internes) ainsi que des valeurs des entrées lues au début du cycle et stockées dans une *mémoire image*. Le programme est en général composé d'expressions Booléennes, ces expressions sont évaluées (calculées) à chaque cycle et leurs résultats sont stockés dans des variables temporaires. Une fois que toutes les expressions sont évaluées de manière séquentielle, les signaux de sorties sont mis à jour afin de commander les actionneurs du système.

Pour terminer sur les notions liées aux SED et aux API, les formalismes et les langages de programmation disponibles sont présentés.

I-3.1.3 Formalismes et langages de programmation

La modélisation permet de mieux maîtriser la complexité du système à étudier, qu'il soit matériel ou immatériel, en passant du monde « réel » au monde « formel ». La modélisation s'appuie sur une abstraction formelle à l'aide d'outils mathématiques appelés

« formalismes ». Il n'existe pas de modèle unique, chaque système doit être considéré comme un cas particulier nécessitant des approches et des formalismes différents.

Dans le cas du développement d'un contrôleur pour un système industriel, différents modèles sont nécessaires : modèle du système physique (capteurs, actionneurs et interactions possibles), modèle des exigences (ce que l'on souhaite faire) et le modèle du contrôleur qui doit permettre la réalisation des exigences en accord avec le système. Ces modèles ne se situent pas au même niveau (matériel, logiciel, verbal). Il n'est donc pas évident de trouver un formalisme mathématique unique permettant de les faire communiquer.

Dans le monde des SED, plusieurs formalismes ont été développés afin de réaliser cette tâche : les automates à états finis (Mealy, 1955; Moore, 1962; Cassandras et Lafortune, 2009), les réseaux de Petri (Petri, 1962; Combacau *et al.*, 2005), le Grafcet (David, 1995), les équations logiques (Hietter, 2009; Marangé, 2008), l'algèbre (max,+) (Lhommeau, 2003)... Chaque formalisme est spécialisé pour un ou plusieurs niveaux de modélisation, sur une certaine catégorie de système ou bien encore avec une représentation mathématique plus ou moins présente. Il dépend du concepteur de trouver le formalisme qui lui conviendra le mieux par rapport à son problème et à ses connaissances.

Quel que soit le modèle (et le formalisme) choisi, celui-ci permet de passer du réel au formel. Cependant, l'opération inverse est également nécessaire, dans le cas d'une conception de contrôleur cette opération est appelée « implémentation ». L'implémentation doit permettre le passage du modèle final du contrôleur exprimé dans un certain formalisme, à un langage de programmation utilisé pour contrôler le système. Ce langage dépend des caractéristiques techniques du matériel (automate programmable industriel, ordinateur, micro-contrôleur...) mais également de la politique technique de l'entreprise et des besoins spécifiques liés au métier.

Il existe de nombreux langages de programmation « classiques » (C, Java, Python...), pour ce qui est des automates programmables industriels il existe 5 langages normalisés de programmation : Structured Text (ST), Ladder Diagram (LD), Instruction List (IL), Function Block Diagram (FBD) et Sequential Function Chart (SFC) (IEC61131-3, 2013). Il est important de préciser que le SFC est un peu particulier, car il utilise les autres langages pour définir les conditions d'évolution du programme. Enfin, certains fournisseurs d'API intègrent des langages de programmation « classiques » tel que le C ou le C++

(B&R automation).

Dans la section suivante nous présentons la méthodologie académique la plus étudiée pour la modélisation et la synthèse d'un contrôleur de SED : la théorie de la commande par supervision.

I-3.2 Théorie de la commande par supervision

La Théorie de la Commande par Supervision (SCT) a été proposée par Ramadge et Wonham (1989) et se base sur le formalisme des langages formels.

La SCT fournit des algorithmes permettant de synthétiser (calculer) un modèle de superviseur à partir du modèle du système à commander et des spécifications (modélisation des exigences). Ces algorithmes calculent automatiquement le superviseur le plus permissif possible (qui restreint le moins le système) garantissant que le comportement du système sous contrôle respecte l'ensemble des spécifications. De nombreux ouvrages traitent de la SCT d'un point de vue théorique (Cassandras et Lafortune, 2009; Wonham, 2015), l'annexe A présente quelques notions fondamentales liées à cette théorie.

La figure 13 présente les étapes de la SCT ainsi que son positionnement par rapport aux étapes de développement d'un contrôleur. A partir de différents modèles, représentés le plus souvent à l'aide d'automates à états finis, la SCT propose des outils mathématiques et algorithmiques permettant la composition de ces modèles et la synthèse automatique du modèle du superviseur. Étant donné que les modèles sont formels, il est possible d'automatiser également les tests à chaque étape de conception via la vérification de propriétés mathématiques. Par soucis de simplicité nous avons choisi de ne pas faire apparaître les tests sur la figure 13.

Il est important de noter que la SCT ne couvre qu'une partie de la conception d'un contrôleur. En effet, l'étape d'analyse, l'extraction du modèle du contrôleur et l'implémentation/intégration ne sont pas traitées dans le cadre formel de la SCT. Les points d'entrées et de sortie de la SCT ne sont que des modèles sous forme de langages, d'automates à états finis ou bien encore de réseaux de Petri.

Même si la SCT a fait ses preuves d'un point de vue formel, il est difficile de l'utiliser d'un

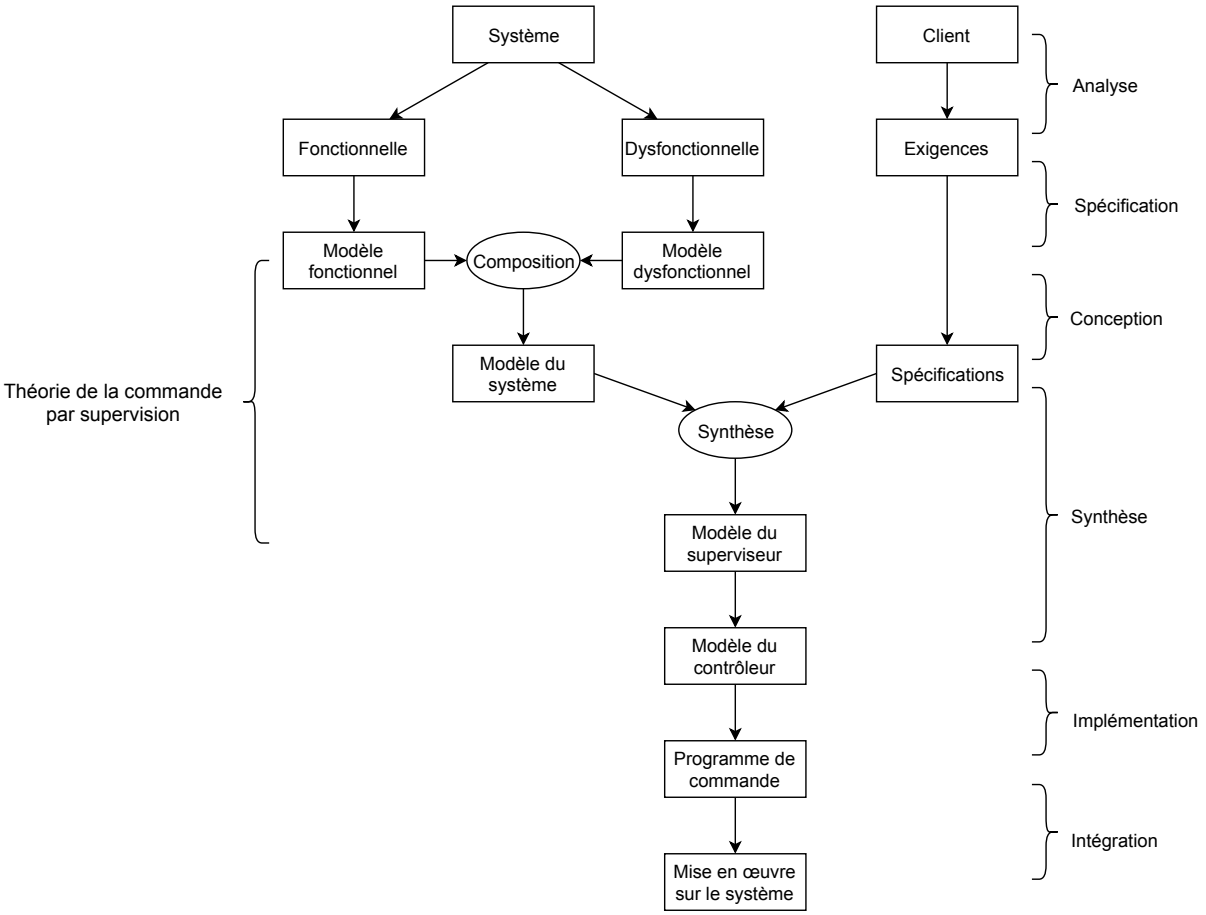


FIGURE 13 – Principe et positionnement de la SCT

point de vue applicatif lorsque l'on souhaite développer un contrôleur (Charbonnier *et al.*, 1999; Vilela et Pena, 2016; Zaytoon et Riera, 2017). Cela est dû à plusieurs problèmes théoriques et pratiques discutés dans la section suivante (section I-3.3).

I-3.3 Limitations de la SCT

Zaytoon et Riera (2017) ont proposé un tour d'horizon des approches de synthèse et d'implémentation pour les contrôleurs logiques, nous proposons de nous appuyer sur leurs propos dans la suite de cette section pour présenter les principales limitations de la SCT.

I-3.3.1 Spécification : comment créer les modèles

Les contributions actuelles montrent un réel potentiel de la SCT à établir des modèles haut-niveau de systèmes complexes (Flordal *et al.*, 2007; Girault *et al.*, 2016; Dallal *et al.*, 2017).

Si l'on souhaite être capable de synthétiser le contrôleur « parfait » il est nécessaire d'avoir des modèles initiaux « parfaits ». Par parfait il faut comprendre le plus précis et complet possible. La complétude du modèle va consister à prendre en compte tous les comportements possibles (souhaités ou non) du système. La précision doit permettre de faire apparaître assez de granularité (de finesse), pour prendre en compte la moindre variation des informations d'entrées et de sorties.

D'après Zaytoon et Riera (2017), il y a actuellement un manque de proposition méthodologique pour l'étape de spécification : comment aider le concepteur à transformer progressivement les exigences (informelles) en spécifications (formelles) ainsi qu'à modéliser un système complexe.

Dans un contexte industriel, les composants élémentaires du système sont la plupart du temps normalisés (vérins, moteurs, capteurs...). Par conséquent, une solution possible, mais peu exploitée de nos jours, est de modéliser chaque composant individuellement puis de créer le modèle global comme étant un réseau de composants interconnectés (Tiller, 2012; Piétrac, 2018). Pour faciliter l'utilisation de cette approche, chaque fabriquant devrait fournir le modèle de son composant dans les différents formalismes utilisés.

Par conséquent, que l'on soit industriel ou académique, de nos jours les modèles sont créés à la main et leur qualité dépend des compétences des concepteurs ainsi que du temps alloué à cette tâche.

I-3.3.2 Synthèse : risque d'explosion combinatoire

Considérons à présent que les modèles (système et spécifications) ont été établis. La SCT nous offre une méthodologie et des algorithmes permettant la synthèse automatique d'un modèle de superviseur. Néanmoins un problème bien connu de la SCT est le risque d'explosion combinatoire. La notion d'explosion combinatoire est liée à la taille du problème ainsi qu'à la complexité des algorithmes permettant de traiter le problème.

Dans le cas des systèmes à événements discrets, l'espace d'état à considérer est discret et sa dimension dépend du nombre de variables : N variables booléennes impliquent un espace d'état de dimension 2^N. Lorsque l'on souhaite modéliser un système manufacturier à l'aide d'un SED, nous associons en général à chaque capteur et actionneur 2 événements : front montant et front descendant. Par conséquent pour un système manufacturier comportant N éléments (capteurs/actionneurs) il faut considérer $2 \times N$ événements et donc une dimension d'espace d'état en 2^{2N}.

Un SED est donc par nature gourmand en quantité de mémoire pour les ordinateurs. De plus, les algorithmes de composition des modèles ont une complexité exponentielle ce qui rend le problème de synthèse d'un superviseur NP-complet (Gohari et Wonham, 2000). Néanmoins, il est à noter que la complexité effective des algorithmes de la SCT dépend de la structure du problème. Par conséquent, il est possible d'appliquer différentes heuristiques de calcul en fonction du problème (Vahidi *et al.*, 2006). Enfin, la SCT a pour but de fournir un superviseur minimal, c'est-à-dire un superviseur le plus petit possible permettant de garantir le fonctionnement du système. Par conséquent, dans certains cas même avec un problème initial comportant des centaines de variables, le superviseur final peut contenir quelques centaines d'états.

I-3.3.3 Implémentation : problèmes liés aux API

Dès lors qu'un modèle de superviseur et/ou de contrôleur a été établi, il est nécessaire de le traduire dans un langage de programmation permettant le contrôle du système réel. Cette étape d'implémentation n'est pas traitée directement dans le cadre de la SCT et de nombreuses discussions à ce sujet ont été proposées dans la littérature (Fabian et Hellgren, 1998; Roth *et al.*, 2010; Zaytoon et Riera, 2017). Nous proposons de résumer les principaux points à prendre en compte lorsque l'on souhaite implémenter un superviseur de la SCT dans un Automate Programmable Industriel (API).

Signaux logiques et événements La SCT est une approche symbolique à base d'**événements** qui apparaissent de façon asynchrone à n'importe quel instant. Les API quant à eux sont des systèmes physiques utilisant des **signaux** (électriques, logiques) et un programme informatique utilisant des **variables** permettant l'acquisition/émission de ces signaux.

Un événement est une information instantanée alors qu'une variable contient une information constante dans un intervalle de temps, par conséquent une variable de l'API ne peut pas correspondre directement à un événement de la SCT. Afin de pouvoir implémenter un superviseur de la SCT dans un API il est donc nécessaire d'effectuer des adaptations au niveau de la méthode ou des modèles liés à la SCT (Piétrac, 2018).

Une approche classique consiste à associer un événement à chaque front (montant et descendant) d'un signal de l'API. Chaque événement pourra alors être associé à une variable dans l'API. Néanmoins cette solution entraîne d'autres problèmes lorsque le système à contrôler émet des signaux trop vite pour que l'API ait le temps de les traiter (Fabian et Hellgren, 1998).

Mémoire, capacité de calcul et expressivité Comme discuté précédemment, un API est en général programmé à l'aide d'un ou plusieurs des 5 langages normalisés (Section I-3.1.3). La question est donc de savoir si un langage est plus adapté qu'un autre à notre problème. Certains langages sont graphiques (LD, SFC, FBD), et d'autres textuels (ST, IL). Certains sont plus adaptés pour exprimer des comportements combinatoires (LD) ou

séquentiels (SFC). Enfin, certains sont au niveau des fonctions (FBD) ou bien au niveau matériel (IL). Le langage de programmation cible est donc dépendant de la nature même du système et du contrôleur, mais également des habitudes et préférences de chacun. En effet, d'après la norme sur ces langages de programmation (IEC61131-3, 2013), ils sont tous équivalents et il est donc normalement possible de passer de l'un à l'autre.

Une fois le langage choisi (ou un assemblage de langage), la question du dimensionnement de l'API doit se poser. Nous avons déjà montré que le superviseur peut comporter de très nombreux états, transitions et événements (Section I-3.3.2), il est donc nécessaire de créer dans l'API de très nombreuses variables et lignes de codes afin de pouvoir traduire correctement le comportement du superviseur. Un API possède un espace mémoire de stockage limité (quelques dizaines de MB pour les plus gros) et un espace de mémoire de travail (RAM) encore plus restreint.

La transformation automatique d'un superviseur dans un langage automate est régulièrement traité dans la littérature (Fabian et Hellgren, 1998; Vieira *et al.*, 2006, 2017). Les langages les plus utilisés sont le LD, le SFC et le ST, néanmoins le passage à l'échelle pour un système industriel complet et complexe reste un problème ouvert.

Temps de cycle et synchronisation Comme nous l'avons présenté précédemment dans la section I-3.1.2, un API a un fonctionnement cyclique avec un temps de cycle constant ou variable. de par cette nature cyclique les signaux d'entrées sont lus uniquement au début d'un cycle, de plus pour l'API les événements provenant du système sont équivalent à des changements sur ces signaux d'entrées.

Ces phénomènes impliquent que, si pendant deux cycles successifs, plusieurs signaux changent, alors les événements associés seront reconnus simultanément quel que soit leur instant d'occurrence respectif. Cet effet de synchronisme des événements est en contradiction avec l'hypothèse d'asynchronisme des événements de la SCT. Des modifications de la SCT doivent dès lors être effectuées pour être robustes à ce phénomène (Fabian et Hellgren, 1998) mais ceci au prix d'une complexité accrue lors de la modélisation.

Initialisation Afin de pouvoir garantir des propriétés structurelles sur les modèles de la SCT (atteignabilité, contrôlabilité...), ainsi que pour pouvoir utiliser le formalisme des

automates à états finis déterministes, un **état initial** unique est nécessaire dans chacun des modèles mis en jeu. Par conséquent, le superviseur résultant de la SCT comportera également un état initial unique.

Dans le contexte de la commande des Systèmes Automatisés de Productions (SAP), la vie de l'entreprise et du système fait que des arrêts (programmés ou non) sont nécessaires. Ces arrêts et reprises du fonctionnement impliquent qu'il peut exister une désynchronisation entre l'état du système et du contrôleur. La prise en compte de ces phénomènes est alors nécessaire dès les phases d'analyse de modélisation.

Il est par exemple possible de rajouter un modèle « haut niveau » (par rapport au contrôleur) permettant la gestion de ces modes de marche et arrêt. Afin d'aider à la mise en place de ce modèle de gestion des modes de marches, l'ADEPA a proposé le GEMMA : Guide d'Étude des Modes de Marche et d'Arrêt (ADEPA, 1981). Le GEMMA est enseigné depuis plusieurs dizaines d'années mais peu de travaux de recherche existent à son sujet (Ponsa *et al.*, 2009). Afin de traiter ces aspects, la communauté scientifique s'est tournée vers une nouvelle classe de problème : la reconfiguration (Mehrabi *et al.*, 2000).

Quelles que soient les méthodes utilisées pour la gestion des modes de marche et d'arrêt, ce problème intervient en partie à cause des formalismes utilisés pour la modélisation : automates à états finis déterministes, réseaux de Petri, GRAFCET. Ces formalismes imposent en effet un état initial. Des approches utilisant d'autres formalismes, à base d'équations logiques, permettent de limiter l'impact de l'état initial (Hietter, 2009; Marangé, 2008).

I-3.3.4 Bilan

D'un point de vue théorique, la SCT permet l'obtention automatique du superviseur le moins restrictif possible à partir d'une modélisation du problème. D'un point de vue pratique, la SCT ne couvre qu'une partie des étapes de conception d'un contrôleur. De plus l'utilisation des API entraine des contraintes supplémentaires rendant plus difficiles la phase de modélisation et les concepts associés à la théorie.

Pour combler ces manques dans le cycle de développement (Fig. 13), pour réduire la difficulté de modélisation, pour diminuer le risque d'explosion combinatoire ou bien encore pour implémenter de façon automatique le superviseur dans un API, plusieurs travaux ont

été menés afin d'étendre ou de modifier la SCT. Nous proposons par la suite (Section I-3.4) de présenter différentes approches traitant plusieurs des points abordés précédemment.

I-3.4 Extensions et améliorations de la SCT

Depuis les travaux originaux sur la SCT (Ramadge et Wonham, 1989), de nombreuses propositions d'extensions et d'améliorations ont été proposées dans la littérature. Nous proposons dans cette section de présenter des approches apportant des solutions aux problèmes présentés précédemment, c'est-à-dire globalement comment la SCT peut-elle être adaptée pour une utilisation pratique.

I-3.4.1 Aide à la spécification

La modélisation (ou spécification) est le point de départ de toute méthode scientifique. Dans le cas du développement d'un contrôleur, cette tâche est réalisée manuellement et demande un effort conséquent. La plupart du temps cette étape est basée sur plusieurs itérations, le but étant de proposer un modèle et de l'analyser formellement afin de le vérifier et le valider. Des méthodologies générales existent pour guider le concepteur : le GEMMA (Ponsa *et al.*, 2009), l'analyse structurée par tâches (Taillard, 2012), etc. Mais globalement, la complétude et l'exactitude du modèle dépend en grande partie de l'expérience et des compétences du concepteur.

Une autre idée est de fournir au concepteur une bibliothèque de composants élémentaires paramétrables vérifiés et validés. La conception revient alors à l'assemblage de composants élémentaires (Philippot, 2006) et la gestion des interactions entre les composants, les produits et les humains (Gouyon *et al.*, 2004). Cette approche de modélisation guidée par les composants plutôt que par l'analyse semble très intéressante mais elle est peu traitée dans la littérature (Zaytoon et Riera, 2017). En effet, selon les besoins et habitudes de chacun, le formalisme utilisé est différent et aucun ne fait l'unanimité. De plus, il existe de très nombreux composants élémentaires dans le monde industriel, la question est de savoir quel doit être le niveau de détail des modèles élémentaires et lesquels choisir.

Cuer *et al.* (2018) ont récemment proposé une approche permettant, à partir des exigences

du cahier des charges, de construire un modèle par renforcement progressif. Les exigences, initialement décrites en langage naturel, sont traduites sous forme de conditions logiques décrites à l'aide d'automates à états finis. Ces équations logiques constituent l'ensemble des spécifications. À partir de ces dernières, les auteurs montrent qu'il est possible de détecter des incohérences entre les exigences et de les corriger. Par la suite, il est possible de synthétiser un superviseur respectant l'ensemble des spécifications corrigées.

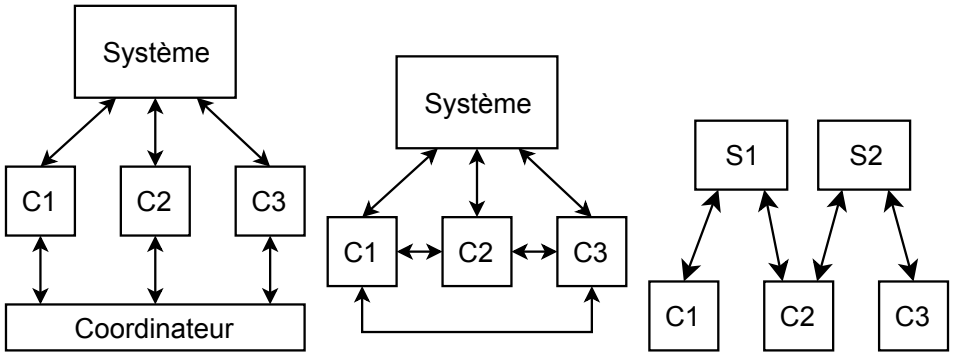
I-3.4.2 Réduction du risque d'explosion combinatoire par l'architecture

L'architecture originale proposée par Ramadge et Wonham peut être qualifiée de centralisée. En effet, à partir d'un modèle unique du système et d'un modèle unique des spécifications, un unique modèle de superviseur est synthétisé. Pour atténuer le risque d'explosion combinatoire lors de l'utilisation de la SCT, différentes architectures de modélisation et de synthèse ont été proposées.

Un système industriel complexe implique de nombreux objectifs à faire cohabiter ensemble dans les contrôleurs : sécurité, maintenance, production... Ces objectifs s'entremêlent et la conception du programme de commande est par conséquent complexifiée. L'approche hiérarchique propose de structurer le programme de commande en plusieurs niveaux, chaque niveau traitant exclusivement d'une partie du problème et communique avec les autres (Wong et Wonham, 1996). L'approche hiérarchique multi-niveaux (Hill *et al.*, 2010) et l'approche de commande organisationnelle généralisent cette idée de hiérarchisation (Seow, 2014).

Il est également possible de considérer plusieurs sous-contrôleurs (contrôleurs locaux) devant travailler en collaboration. Chaque contrôleur local peut observer tout ou partie du système à contrôler, et considérer des sous parties du cahier des charges (disjointes ou non). Plusieurs approches existent en fonction de la façon dont les informations sont transmises et du degré d'autonomie de chaque contrôleur. La commande décentralisée considère 1 unique système et plusieurs contrôleurs locaux. Chaque contrôleur local observe tout ou partie du système pour prendre une décision locale, ensuite un coordinateur récupère les choix fait par chaque contrôleur local et prend la décision finale (Fig. 14a) (Yoo et Lafortune, 2002). L'approche modulaire considère elle aussi 1 unique système

et plusieurs contrôleurs locaux. Dans l'approche modulaire il n'y a pas de coordinateur, les contrôleurs locaux doivent donc communiquer directement entre eux pour obtenir la solution finale (Fig. 14b) (Cassandras et Lafortune, 2009). Enfin l'approche distribuée considère le système global comme étant plusieurs sous-systèmes, les contrôleurs locaux peuvent observer 1 ou plusieurs sous-systèmes (Fig. 14c) (Cai et Wonham, 2010). Il est à noter qu'en fonction des auteurs, la définition formelle de chacune de ces approches varie.



(a) Décentralisée (b) Modulaire (c) Distribuée

FIGURE 14 – Architectures de contrôle

I-3.4.3 Modification des algorithmes

Pour permettre de traiter des problèmes complexes avec la SCT, il est également possible de rendre plus efficace les algorithmes de composition des modèles et de synthèse du superviseur.

L'approche compositionelle est basée sur la construction d'un ensemble de petits automates équivalents à la représentation monolithique (centralisée). Avec cet ensemble, le superviseur est calculé à l'aide d'un espace d'état à explorer plus petit qu'avec l'approche centralisée (Mohajerani *et al.*, 2014).

L'approche symbolique propose d'utiliser la structure des arbres de décision binaire (BDD), ceci dans le but de calculer efficacement le superviseur. De plus, avec cette approche, le superviseur n'est pas défini à l'aide d'automates à états finis mais, à l'aide d'équations logiques (Fei *et al.*, 2014).

I-3.4.4 Prise en compte du fonctionnement de l'API

En général, les restrictions liées à la technologie du contrôleur physique ne sont pas prises en compte lors de la modélisation et de la synthèse d'un contrôleur. Cela peut entraîner de nombreux problèmes comme nous l'avons vu précédemment (Section I-3.3.3). Certains auteurs proposent d'intégrer les comportements des contrôleurs physiques dès la phase de modélisation.

Dans l'approche proposée par Cantarelli et Roussel (2008), un modèle des interactions entre la partie commande et la partie opérative est ajoutée à la phase de modélisation. Ce modèle d'interaction est basé sur plusieurs petits modèles génériques permettant de prendre en compte des changements simultanés des entrées et des sorties. Les auteurs ont également montré qu'en prenant en compte ces interactions, la génération automatique du programme API est simplifiée.

Les problèmes d'implémentation sont principalement liés au comportement cyclique des contrôleurs physiques (lecture, exécution, écriture) (Fabian et Hellgren, 1998). Pour prendre en compte ce phénomène, différentes techniques existent (Mader, 2000). En règle général, la prise en compte du temps dans les modèles permet l'obtention d'un comportement final proche de la réalité (Hanisch *et al.*, 1997; Fabian et Hellgren, 1998; Wang *et al.*, 2011). Néanmoins, ces approches temporelles peuvent sortir du cadre formel de la SCT. L'obtention d'un résultat optimal n'est donc pas obligatoirement garantie.

I-3.4.5 Gestion de l'initialisation par la reconfiguration

Afin de gérer les problèmes liés à l'initialisation des modèles et la synchronisation d'états avec le système physique, plusieurs approches ont été proposées et peuvent être regroupées avec la notion de reconfiguration (Mehrabi *et al.*, 2000). La reconfiguration consiste à adapter automatiquement le programme en fonction de stimuli extérieur, soit l'état « actif » est modifié dans un modèle unique, soit le modèle « actif » est changé en vérifiant que le changement est possible.

Ce domaine est assez récent et par conséquent il existe de nombreuses variantes à la notion de reconfiguration : le système physique peut être reconfigurable et le modèle associé doit

alors le prendre en compte (Zhang *et al.*, 2015), le système physique peut être fixe mais le « produit » traité par le système peut être reconfigurable (Gouyon *et al.*, 2007), ou bien le contrôleur est reconfigurable pendant le fonctionnement du système permettant des changements de trajectoire de la loi de commande (Macktoobian et Wonham, 2017).

I-3.5 Bilan

La théorie de la commande par supervision a radicalement changé la vision de la commande des SED. D'un point de vue formel, cette approche est efficace et élégante, néanmoins d'un point de vue applicatif, cette approche se heurte à d'importants problèmes. Dès lors, de nombreux travaux tentent d'améliorer et d'étendre l'approche originelle dans le but de pouvoir résoudre tout ou partie de ces problèmes.

Néanmoins, les hypothèses fondatrices (asynchronisme et état initial) restent nécessaires dans ces extensions et continuent de limiter l'application sur les systèmes réels. D'autres travaux ont fait le choix radical de repartir du début en proposant de nouvelles approches utilisant des formalismes différents. Dans la suite de ce chapitre, deux approches approches non-basées sur la SCT sont présentées.

I-4 Approches de commande des SED par contraintes logiques

De par la difficulté de modélisation et d'implémentation lorsque la cible finale est un API, des chercheurs ont fait le choix de proposer des approches ne se basant pas sur la SCT et la théorie des langages. Dans le but d'être plus proche des restrictions imposées par l'utilisation des API, et des méthodes de travail de l'automaticien, les approches présentées dans cette section se basent sur la notion de contraintes logiques utilisant des variables logiques (Booléennes).

I-4.1 Algèbre de Boole : définitions et notations

Cette section a pour objectif de poser les définitions et notations, utilisées par la suite dans le manuscrit, liées à l'algèbre de Boole.

Définition 3. ***Algèbre de Boole*** *(d'après (Hietter, 2009))*

Soit $\mathbb{B}$ un ensemble non vide d'éléments contenant deux éléments particuliers 0 (élément Zéro) et 1 (élément Un), sur lequel sont définies deux lois de composition interne binaires notées $(+, \cdot)$, et une loi de composition interne unaire notée $(\,\overline{}\,)$.

$(\mathbb{B}, +, \cdot, \overline{}, 0, 1)$ est une algèbre de Boole si les neuf axiomes suivants sont satisfaits pour tout élément x, y et z de $\mathbb{B}$:

$$x + y = y + x \quad [1] \qquad x \cdot y = y \cdot x \quad [2]$$

$$x + (y \cdot z) = (x + y) \cdot (x + z) \quad [3] \qquad x \cdot (y + z) = (x \cdot y) + (x \cdot z) \quad [4]$$

$$x + 0 = x \quad [5] \qquad x \cdot 1 = x \quad [6]$$

$$x + \overline{x} = 1 \quad [7] \qquad x \cdot \overline{x} = 0 \quad [8]$$

$$0 \neq 1 \quad [9]$$

L'algèbre de Boole historique est celle des variables booléennes (Boole, 1854). Néanmoins différentes algèbres de Boole peuvent être définies. Les travaux de Hietter ont notamment étendus les résultats de l'algèbre de Boole classique à l'algèbre des fonctions booléennes (Hietter, 2009).

Définition 4. ***Variables booléennes*** *(ou variables logiques)*

Une variable x est appelée variable booléenne si elle ne peut prendre comme valeur que 0 ou 1.

Dans ce manuscrit, les trois lois de composition interne $(+, \cdot, \overline{})$ sont nommées respectivement ET, OU, NON. Soient x et y deux variables booléennes, alors les tables de vérité de ces lois sont présentées ci-dessous :

Par analogie à la somme et au produit de plusieurs nombres réels ($\sum$ et $\prod$), il est possible de définir une somme logique (Déf. 5) et un produit logique (Déf. 6) sur des variables booléennes.

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

(a) ET

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

(b) OU

x	$\overline{x}$
0	1
1	0

(c) NON

TABLE 1 – Lois de composition interne de l'algèbre de Boole

Définition 5. ***Somme logique***

Soient V un ensemble non vide de variables booléennes et $v_i \in V$ la i-ème variable booléenne de V. Alors la somme logique des variables booléennes de V est notée $\sum V$ et définie ci-dessous :

$$\sum V = \sum_{i=1}^{dim(V)} v_i = v_1 + v_2 + \ldots + v_{dim(V)}$$

Définition 6. ***Produit logique***

Soient V un ensemble non vide de variables booléennes et $v_i \in V$ la i-ème variable booléenne de V. Alors le produit logique des variables booléennes de V est notée $\prod V$ et définie ci-dessous :

$$\prod V = \prod_{i=1}^{dim(V)} v_i = v_1 \cdot v_2 \cdot \ldots \cdot v_{dim(V)}$$

Ces notations seront utilisées dans la suite du manuscrit.

I-4.2 Synthèse algébrique

Hietter (2009) a proposé une approche de synthèse algébrique d'une loi de commande pour les SED. La synthèse algébrique est basée sur le formalisme de l'algèbre de Boole et la résolution d'un système d'équations Booléennes. Dans ses travaux de thèse, Hietter a proposé et démontré des théorèmes et algorithmes permettant de résoudre un système d'équations Booléennes de taille quelconque. La méthode proposée est basée sur 4 étapes (Fig. 15).

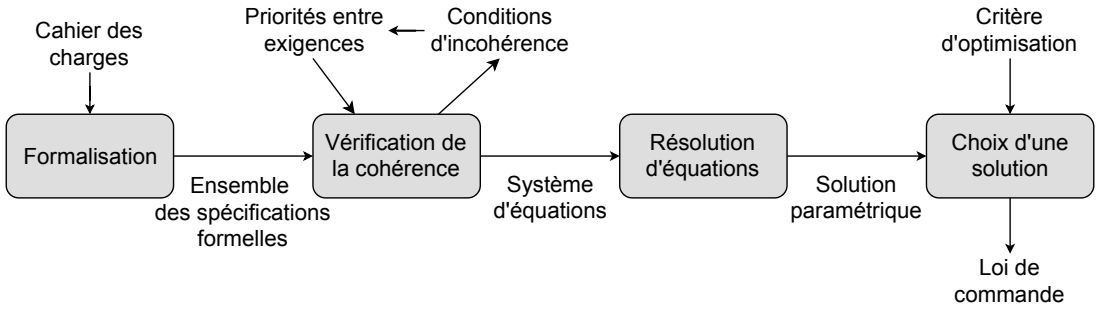


FIGURE 15 – Méthodologie de la synthèse algébrique

La phase de formalisation permet de traduire les exigences, données en langage naturel, vers des spécifications sous forme d'équations logiques. Les variables constituant ces équations logiques peuvent être *connues* ou *inconnues*. Une fois les exigences formalisées, une unique équation canonique est créée à partir des spécifications logiques. Cette équation va être analysée à l'aide d'un test de cohérence. Si les spécifications ne sont pas cohérentes entre elles, un contre-exemple est donné. Ce contre-exemple peut alors être utilisé pour définir des priorités entre différentes équations logiques ou bien pour corriger certaines spécifications (et donc reprendre les exigences). Une formalisation de cette partie de l'approche est détaillée dans Roussel et Lesage (2014). Si l'ensemble est cohérent, une équation paramétrique de chaque variable *inconnue* est alors calculée. Enfin, à l'aide de critères d'optimisation propres à chaque application, il est nécessaire de choisir la valeur du paramètre constituant la solution paramétrique.

Hietter a montré dans ses travaux qu'il était possible de traduire un ensemble d'exigences, écrites en langage naturel, vers un ensemble de spécifications sous formes d'équations logiques (Tab. 2).

TABLE 2 – Formalisation de phrases en langage naturel (Hietter, 2009)

Type	Structure de phrase type en langage naturel	Formalisation logique
Condition suffisante	Si A est vrai alors B est également vrai	$A \Rightarrow B$ $A \cdot \overline{B} = 0$
	Lorsque A est présent, B est présent	
	Il suffit d'avoir A pour avoir B	
Condition nécessaire	Il est nécessaire d'avoir A pour avoir B	$B \Rightarrow A$ $B \cdot \overline{A} = 0$
	A est obligatoire pour avoir B	
	A autorise d'avoir B	
Situation interdite	Il est interdit d'avoir A et B simultanément	$A \cdot B = 0$
	A ne peut pas être actif en même temps que B	
Invariant	Il faut toujours avoir A ou B	$A + B = 1$

Ces équations logiques sont composées de variables *connues* et *inconnues*. Les variables connues vont représenter les états de capteurs, les états de mémoires et autres variables internes constantes durant un cycle automate. Les variables inconnues quant à elles vont représenter les états des actionneurs.

A partir de cet ensemble d'équations logiques, le but est de trouver une équation pour chaque variable inconnue en fonction des variables connues. Ces solutions peuvent ne pas exister, une condition d'existence a été démontrée. Si les solutions existent, elles peuvent être représentées sous forme paramétrique ou non (tout comme dans une résolution d'équations dans $\mathbb{R}$). Si la solution est paramétrique, la solution finale est obtenue en choisissant une valeur pour chaque paramètre.

Étant donné que la solution pour chaque variable d'actionneur est une équation logique, il est très simple de l'implémenter automatiquement dans un API. Par conséquent, à condition que l'écriture du cahier des charges suivent certaines règles syntaxiques, la synthèse algébrique proposent une méthode quasiment tout automatique pour implémenter une loi de commande dans un API à partir du cahier des charges. La seule opération manuelle est le choix de la valeur des paramètres dans la solution finale.

Cependant, dans un contexte industriel cette approche pose plusieurs difficultés. En effet, il existe 3 principales limitations à l'utilisation de la synthèse algébrique :

— Explosion combinatoire : l'algorithme de recherche de solution paramétrique a un fort risque d'explosion combinatoire. La complexité de l'algorithme dépend du nombre de variables et de contraintes, dans un système industriel il peut exister plusieurs centaines de variables formant plusieurs centaines de contraintes.
— Choix du paramètre : si la solution est paramétrique il est nécessaire de choisir ce paramètre, néanmoins il semble très difficile d'associer une signification physique ou logique à la valeur d'un paramètre dans le cas général. Ce choix de paramètre est donc arbitraire, pourtant il n'est pas anodin : deux choix différents impliquent deux solutions différentes, donc potentiellement deux comportements différents du système.
— Lisibilité de la solution : à chaque variable d'actionneur est associée une équation logique, mais celle-ci peut comporter plusieurs centaines de variables logiques. Il semble donc difficile de maintenir le code ou de comprendre les équations finales.

Ceci pose donc un problème lors de l'exploitation et de la maintenance d'un système contrôlé par une telle loi de commande.

La synthèse algébrique apporte donc des outils mathématiques très intéressants pour la commande d'un SED à base de contraintes logiques. Néanmoins elle nécessite de considérer en même temps les notions d'objectifs (ce que l'on veut faire) et de contraintes (ce que l'on ne doit pas faire). Cette séparation des notions est importante pour réussir à appréhender sereinement un système conséquent (Charbonnier *et al.*, 1999; Cuer *et al.*, 2018).

I-4.3 Filtre logique de commande

I-4.3.1 Principe d'un filtre et utilisations

Dans le but de pouvoir séparer efficacement les différentes notions intervenant dans un programme de commande, la notion de filtre a été proposée (Cruette *et al.*, 1991). Un filtre est un élément (logiciel ou matériel) placé entre la partie opérative (système physique) et la partie commande (Fig. 16). Ce filtre permet, d'une part de s'assurer que les commandes émises soient bonnes, et d'autre part que le retour de la partie opérative correspond à un fonctionnement attendu (détection de défauts). Dans ces travaux, le filtre est modélisé à l'aide de réseaux de Petri.

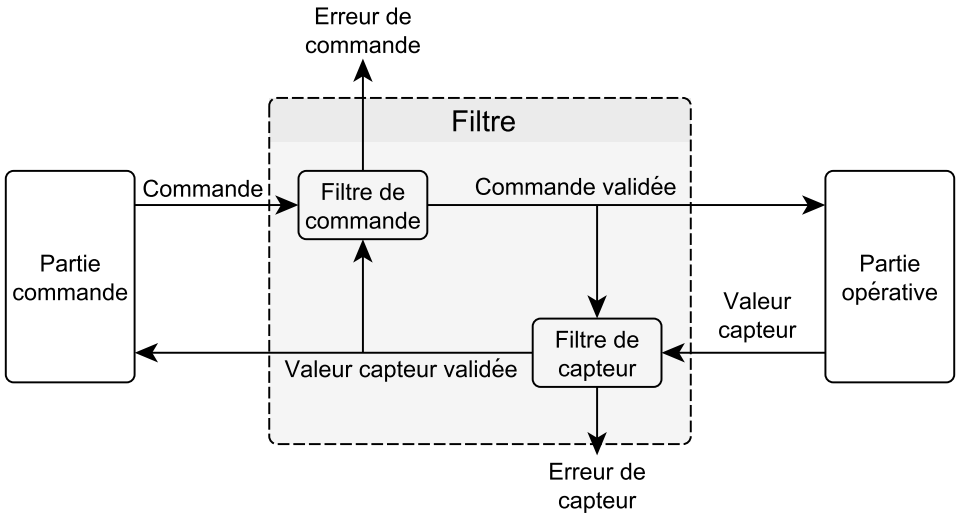


FIGURE 16 – Principe d'un filtre

En se basant sur cette notion de filtre, Marangé (2008) l'a adaptée pour proposer un filtre

de sécurité en se basant sur des contraintes logiques. Dans l'approche de Marangé, seule la partie commande du filtre est traitée, la partie opérative est considérée comme sûre de fonctionnement. Dans cette approche, toutes les contraintes de sécurité sont fixées par un expert du système qui définit les états interdits du système et également les séquences à interdire. Dès qu'une contrainte de sécurité est violée, le filtre change la commande et bloque le système dans un état de repli prédéterminé.

Ces travaux (Cruette *et al.*, 1991; Marangé, 2008) étaient appliqués dans un cadre de commande sûre de fonctionnement, et de détection d'erreurs de la partie commande. Des travaux dans ce sens ont continué d'être développés au CReSTIC et sont présentés dans la section I-4.3.2. Il est à noter que des applications liées à la cybersécurité sont par ailleurs actuellement étudiés par la communauté scientifique :

— Sicard *et al.* (2017) se base sur la notion de filtre, mais utilise une notion de distance entre états pour la détection d'une cyberattaque. Une attaque est détectée lorsque l'état actuel de la partie commande s'éloigne trop de la trajectoire attendue des états.
— Toublanc *et al.* (2017) spécialise la notion de filtre en définissant des *gardiens*. Trois gardiens différents sont définis, chacun devant détecter une erreur de la partie commande, de la partie opérative ou du réseau de communication. Ces gardiens sont organisés dans une passerelle entre la partie commande (PC) et la partie opérative (PO), cette passerelle est positionnée au niveau de la communication entre la PO et la PC.

I-4.3.2 Travaux relatifs au filtre de commande

Plusieurs travaux ont étendu et complété les travaux de Marangé (2008), cette partie présente les différentes évolutions de ces travaux.

Initialement, le filtre de commande était constitué d'un ensemble de contraintes logiques appelées contraintes de sécurité (Marangé *et al.*, 2007). Deux types de contrainte étaient définis :

— les contraintes *statiques* : faisant apparaître uniquement des variables ;
— les contraintes *dynamiques* : faisant apparaître des événements (fronts d'activation

et de désactivation de variables).

Ces contraintes étaient implémentées de manière séquentielle, à la fin du cycle automate, avant la mise à jour des sorties (Fig. 17). Si au moins une contrainte était violée (évaluée à vraie), alors les valeurs des sorties étaient modifiées pour atteindre un état de repli prédéterminé et stable. Ce filtre peut donc être qualifié de **bloquant**.

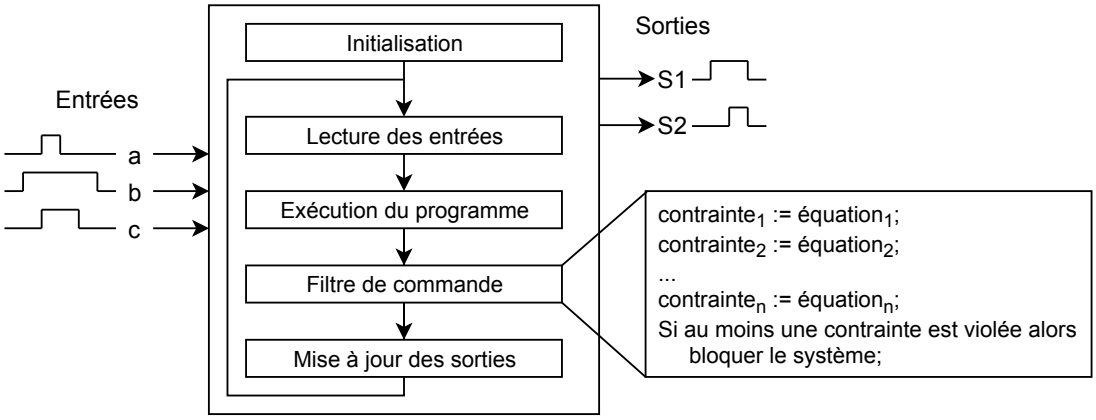


FIGURE 17 – Principe d'implémentation du filtre bloquant dans un API

Marangé a également proposé une approche hors-ligne à base de model-checking permettant de valider un ensemble de contraintes logiques par rapport aux exigences (Fig. 18). En se basant sur une modélisation de la partie opérative à l'aide d'automates à états finis, ainsi que sur une liste d'états et de comportements interdits, il est possible de vérifier par model-checking la propriété de **suffisance** : quelle que soit la valeur des capteurs et le programme précédent le filtre, l'ensemble de contraintes est suffisant pour garantir qu'aucun état interdit ou comportement interdit ne soit atteignable. Dans le cas où la propriété n'est pas vérifiée, un contre-exemple est remonté pour aider le concepteur du filtre.

L'approche par filtre bloquant permet la vérification formelle hors-ligne de programme automate existant, ainsi que la détection en-ligne d'erreurs provenant de la partie commande. Néanmoins, le blocage systématique du système lors de la détection d'une erreur peut entrainer des arrêts de production trop fréquents. Afin de diminuer le risque de blocage du système, Benlorhfar *et al.* (2011) ont proposé la notion de filtre **correcteur**. Le principe est de modifier l'implémentation du filtre afin de proposer en-ligne des valeurs de sortie alternative, basées sur les valeurs envoyées par la partie commande, qui ne violent aucune contrainte de sécurité (Fig. 19). Dans ces conditions, le système n'est pas forcément bloqué et peut potentiellement continuer la production dans un mode dégradé.

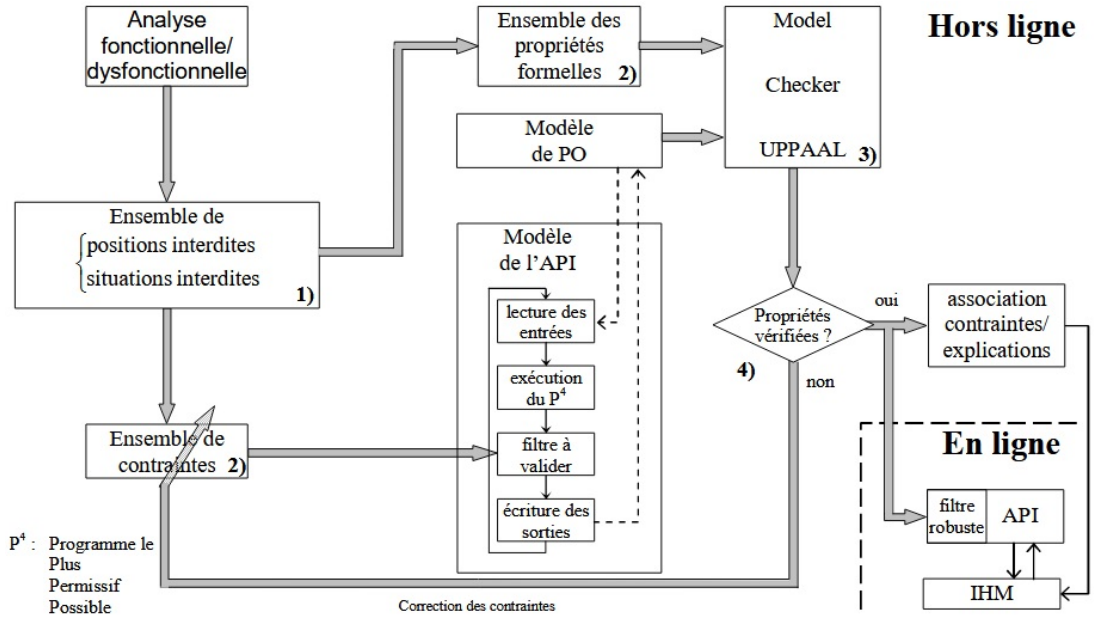


FIGURE 18 – Approche par model-checking de vérification de la suffisance (Marangé, 2008)

Riera *et al.* (2015) ont montré que le filtre correcteur pouvait être utilisé de deux façons différentes : « filtre correcteur superviseur » et « filtre correcteur contrôleur ». Dans le premier cas, le programme fonctionnel est réalisé sans avoir connaissance de l'existence du filtre correcteur en aval. Dans le second cas, la connaissance du filtre est prise en compte dans la conception du programme fonctionnel. En d'autres termes, dans le cas du filtre correcteur contrôleur, violer une contrainte peut être considéré comme un fonctionnement « normal » du filtre. A contrario, dans le cas du filtre correcteur superviseur, la violation d'une contrainte signifie que le programme fonctionnel comporte au moins une erreur de conception.

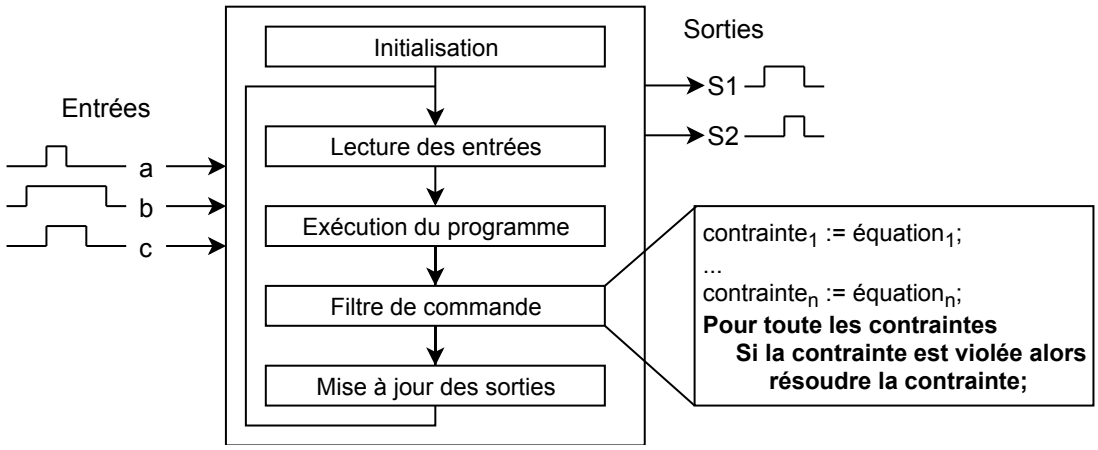


FIGURE 19 – Principe d'implémentation du filtre correcteur dans un API

Avec le filtre correcteur il est donc possible de modifier dynamiquement la commande

sans pour autant forcement bloquer le système. Néanmoins avec l'implémentation séquentielle proposée dans Benlorhfar *et al.* (2011), la résolution d'une contrainte peut violer une contrainte précédemment non-violée ou résolue. De plus, en fonction de l'ordre dans lequel les contraintes sont implémentées, la solution n'est pas obligatoirement la même, la résolution n'est donc pas déterministe.

Afin de résoudre ces problèmes, Coupat (2014) a proposé un algorithme itératif permettant la résolution déterministe d'un ensemble de contraintes de sécurité. Un algorithme succin est présenté Fig. 20, l'algorithme détaillé est présenté dans Coupat (2014) (Fig. 55, p. 141). Enfin, dans les travaux de Coupat, une redéfinition des contraintes a été proposée dans le but d'étendre l'approche par filtre à la commande sûre de fonctionnement. Les deux nouveaux types de contraintes, remplaçant ceux définis par Marangé, sont les suivants :

— les contraintes **simples** : ne contiennent qu'une seule variable de sortie ;

— les contraintes **combinées** : contiennent plusieurs variables de sorties.

En plus de ces différences structurelles, Coupat propose d'utiliser ces contraintes pour deux notions différentes : les contraintes de **sécurité** et les contraintes **fonctionnelles**. Les premières ont pour objectif de formaliser les exigences de sécurité du cahier des charges (configurations interdites du système), les secondes ont pour objectif de formaliser la partie fonctionnelle (configurations souhaitées du système).

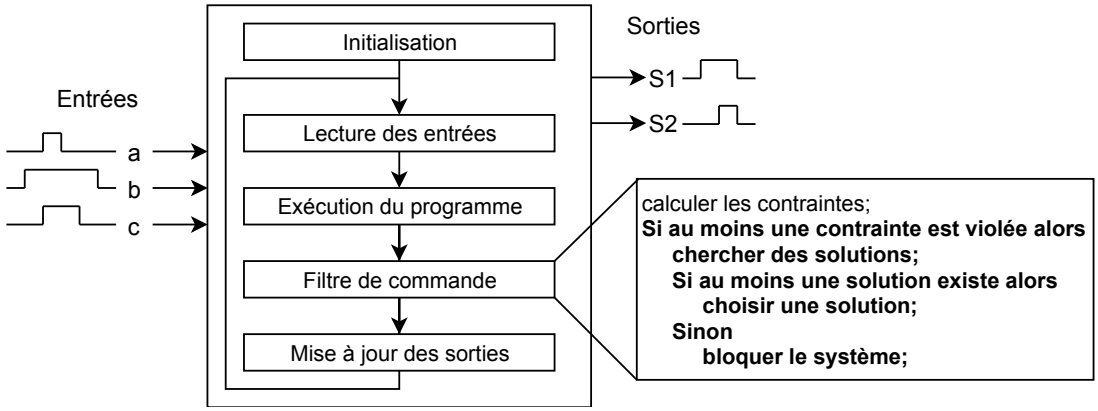


FIGURE 20 – Principe de l'algorithme itératif

Les travaux de Coupat ont permis une première application industrielle de l'approche par filtre de commande avec la SNCF. Le problème était de garantir que des programmes API existants étaient corrects du point de vue de la sécurité, dans le cas contraire, la sécurité devait être garantie en n'effectuant aucune modification sur le programme API existant. Coupat a proposé une approche à base de model-checking pour tester les programmes API.

Afin de mettre en sécurité un programme « erroné », l'approche par filtre de commande a été proposée. Coupat a finalement montré par model-checking que dans le cas d'un programme initial « erroné », l'association d'un filtre de commande au programme permet de garantir la sécurité et de garantir un fonctionnement correct du contrôleur.

Dans les travaux de Marangé une propriété de suffisance de l'ensemble des contraintes a été proposée. Dans l'algorithme proposé par Coupat (Fig. 20), s'il n'existe pas de solution lors de la résolution, alors une solution arbitraire va être appliquée, mais cette solution ne vérifie pas l'ensemble des contraintes. L'expression d'une propriété d'existence de solution est donc nécessaire.

La notion de **cohérence** d'un ensemble de contraintes de sécurité a donc été proposée (Riera *et al.*, 2015). Un ensemble de contraintes de sécurité est cohérent si et seulement si, quelle que soit la valeur des entrées du filtre (capteurs, variables internes, commandes envoyées par le programme amont), il existe toujours une solution (valeurs des commandes) vérifiant l'ensemble des contraintes. Riera *et al.* (2015) ont également proposé un ensemble de conditions nécessaires à la cohérence d'un ensemble de contraintes de sécurité.

I-4.3.3 Verrous restants

Après les derniers travaux (Riera *et al.*, 2015), différents verrous restent à lever.

- Modélisation : les contraintes de sécurité doivent être écrites à la main par un expert du système. La traduction du cahier des charges en contraintes peut donc s'avérer compliquée et non exhaustive.
- Vérification : la vérification d'un ensemble de contraintes (propriété de cohérence) est basée sur différentes propriétés nécessaires mais non suffisantes. La cohérence ne peut donc pas être garantie.
- Algorithme de résolution : l'implémentation de l'algorithme proposé (Coupat, 2014) restreint l'expressivité des contraintes. En effet, les contraintes ne doivent contenir que 2 variables de sorties au maximum. Ceci impacte l'expressivité des contraintes et la possibilité de modélisation.

I-4.4 Liens avec la programmation par contraintes

La programmation par contraintes (PPC, ou CP pour *constraint programming*) est un paradigme de programmation apparu entre les années 1970 et 1980 (Haralick et Elliott, 1980; Mackworth, 1981) permettant de résoudre des problèmes combinatoires de grande taille tels que les problèmes de planification et d'ordonnancement (Baptiste *et al.*, 2012). En programmation par contraintes, on sépare la partie modélisation à l'aide de problèmes de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*), de la partie résolution dont la particularité réside dans l'utilisation active des contraintes du problème pour réduire la taille de l'espace des solutions à parcourir.

Dans le cadre de la programmation par contraintes, les problèmes sont modélisés à l'aide de variables de décision et de contraintes, où une contrainte est une relation entre une ou plusieurs variables qui limite les valeurs que peuvent prendre simultanément chacune des variables liées par la contrainte (Déf. 7).

L'approche par filtre logique (Riera *et al.*, 2015) repose également sur la définition de contraintes logiques, il semble donc possible d'utiliser des techniques de PPC pour résoudre des problèmes de commande des systèmes automatisés de production.

Définition 7. ***Problème de Satisfaction de Contraintes (CSP)***

Un problème de satisfaction de contraintes sur des domaines finis (ou CSP) est défini par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- *$\mathcal{X}$ est l'ensemble des variables du problème ;*
- *$\mathcal{D}$ est l'ensemble des domaines des variables, c'est-à-dire les valeurs que peuvent prendre les variables ;*
- *$\mathcal{C}$ est l'ensemble de contraintes sur les variables.*

Dans le cas général, les contraintes utilisent des opérateurs arithmétiques et/ou logiques sur les variables. De plus, les variables peuvent être de n'importe quel type (entières, réelles, logiques...). Une classe particulière de CSP est le **problème de satisfaisabilité booléenne** (problème SAT), qui considère uniquement des variables logiques (Déf. 8). Dans le cas d'un problème SAT, les contraintes sont construites à partir de variables propositionnelles et des connecteurs booléens « et » (conjonction), « ou » (disjonction),

« non » (négation). Les notations classiques pour ces connecteurs sont : $\wedge$ (conjonction), $\vee$ (disjonction), $\neg$ (négation). Dans cette thèse, les mêmes notations que dans les chapitres précédents seront utilisées pour ces connecteurs.

Définition 8. ***Problème SAT***

Un problème de satisfaisabilité booléenne (problème SAT) est défini par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- *$\mathcal{X}$ est l'ensemble des variables propositionnelles du problème ;*
- *$\mathcal{D} = \{true, false\}$ est l'unique domaine des variables ;*
- *$\mathcal{C}$ est un ensemble de contraintes sur les variables.*

La notion de *littéral* est utilisé pour les définitions formelles des contraintes d'un problème SAT. Pour rappel, Un littéral ℓ est une variable propositionnelle v_j (littéral positif) ou la négation d'une variable propositionnelle $\overline{v_j}$ (littéral négatif).

En fonction de la structure des contraintes (connecteurs utilisés, nombres de variables...), différents types de problèmes SAT peuvent être identifiés (CNF-SAT, DNF-SAT, 2-SAT...). Le problème CNF-SAT est le plus couramment utilisé, celui-ci est la restriction du problème SAT aux contraintes écrites sous formes normales conjonctives (CNF).

Définition 9. ***Problème SAT-CNF***

Un problème SAT est en forme normale conjonctive si elle est une conjonction de disjonctions de littéraux ℓ_j.

$$\prod_{i=1}^{dim(C)} \sum_{j} \ell_j$$

Dans un SAT-CNF, les contraintes sont donc des disjonctions de littéraux. Il est également possible de définir un problème utilisant des conjonctions pour définir les contraintes (Déf. 10).

Définition 10. ***Problème SAT-DNF***

Un problème SAT est en Forme Normale Disjonctive (DNF) si elle est une disjonction de conjonctions de littéraux ℓ_j.

$$\sum_{i=1}^{dim(C)} \prod_{j} \ell_j$$

Quelle que soit la structure des contraintes, et les connecteurs utilisés pour lier ces contraintes, le but est ensuite d'avoir des algorithmes capables de statuer sur la satisfaisabilité (ou satisfiabilité) du problème SAT (Déf. 11).

Définition 11. ***Problème satisfaisable***

Un problème SAT est dit ***satisfaisable*** *s'il existe une affectation des variables propositionnelles qui rend la formule logiquement vraie.*

$$\text{SAT-CNF satisfaisable} \Leftrightarrow \prod_{i=1}^{dim(C)} \sum_{j} \ell_j = 1$$

$$\text{SAT-DNF satisfaisable} \Leftrightarrow \sum_{i=1}^{dim(C)} \prod_{j} \ell_j = 1$$

Les algorithmes de recherche de solutions s'appuient généralement sur la propagation de contraintes, pour réduire le nombre de solutions candidates à explorer, ainsi que sur des techniques de *back-tracking*, pour annuler des choix entrainant des conflits entre contraintes. De tels algorithmes garantissent de trouver une solution, si elle existe, et permettent de prouver qu'il n'existe pas de solution à un problème s'ils n'ont pas trouvé de solution à la fin de la recherche exhaustive (Du *et al.*, 1997; Gu *et al.*, 1999).

Il existe dans la littérature de nombreux solveurs SAT implémentant différents algorithmes de résolution (Moskewicz *et al.*, 2001; Eén et Sörensson, 2003; Jussien *et al.*, 2008). Un solveur a pour but de trouver une solution, c'est-à-dire une affectation totale des variables, permettant de satisfaire l'ensemble des contraintes. Si aucune solution n'est trouvée, alors le problème n'est pas satisfaisable. Il est également possible d'obtenir toutes les solutions du problème.

Afin d'optimiser la recherche d'une solution, les solveurs SAT s'appuient sur différentes

techniques algorithmiques. L'algorithme de base d'un solveur SAT est DPLL (Davis *et al.*, 1962). DPLL est un algorithme de recherche arborescent. Dans sa forme initiale, il intègre notamment du back-tracking et de la propagation unitaire. Par la suite, différentes améliorations ont été proposées en termes de techniques, d'heuristiques et de structures de données.

Quelques techniques sont présentées dans la suite de cette section. Des détails sur ces techniques sont disponibles dans la littérature (Du *et al.*, 1997; Gu *et al.*, 1999).

Propagation des contraintes unitaires Une contrainte est unitaire lorsque tous ses littéraux sont affectés sauf un. L'identification de ces contraintes unitaires est effectuée avant et pendant la résolution.

Après avoir choisi une valeur à affecter à un littéral, si une contrainte devient unitaire, le choix pour résoudre cette contrainte est immédiat (un seul littéral est affectable dans la contrainte). Aucun choix n'est donc à faire et la résolution est déterministe. En pratique, résoudre une contrainte unitaire entraine souvent une cascade d'autres clauses unitaires à résoudre. Cela évite donc l'exploration d'une grande partie de l'espace de recherche. Ce mécanisme de cascade est appelé propagation de contraintes unitaires.

Back-tracking Le principe du back-tracking est de revenir sur des décisions prises lors de la recherche de solution, afin de sortir d'un blocage. Il est à noter que, si un conflit est détecté pendant une propagation de contraintes unitaires, la décision à remettre en question n'est pas la dernière affectation effectuée, mais le choix d'affectation ayant entrainé la cascade de résolution unitaire. La gestion du back-tracking implique donc une structure de données, et une capacité de mémorisation suffisante, pour mémoriser les différents choix et implications à « défaire » lors de la détection d'un conflit.

Apprentissage de contraintes par conflits Le principe est d'apprendre de nouvelles contraintes, pendant la recherche, afin de réduire l'espace à explorer. Lorsqu'un conflit est détecté, et avant le back-tracking, une contrainte est créée permettant de traduire le conflit, évitant de le répéter durant la suite de la recherche.

Calcul de l'ensemble des solutions De base, un solveur SAT a pour but de trouver une solution. Il est néanmoins possible de trouver l'ensemble des solutions. Une méthode simple pour cela est d'ajouter la solution calculée, sous forme complémentée, au problème puis de relancer l'algorithme. Dans ces conditions, une nouvelle solution, différente de la précédente, sera calculée.

Les recherches scientifiques sur les problèmes SAT se focalisent sur le développement de solveurs de plus en plus rapides et capables de traiter des problèmes toujours plus grands. En effet, les instances (problèmes) utilisées lors des compétitions comportent des dizaines de milliers de variables pour des centaines de milliers de contraintes (Balyo *et al.*, 2017). Enfin, ces solveurs sont faits pour être exécutés sur des ordinateurs « classiques », et les temps de calcul peuvent atteindre plusieurs heures.

Si l'on considère un système automatisé de production, celui-ci comporte en général quelques centaines de variables au maximum. De plus, si un tel système est considéré comme un système à événements discrets, il est possible de modéliser son comportement à l'aide d'équations logiques (Hietter, 2009; Riera *et al.*, 2015). L'utilisation des techniques et solveurs SAT semble alors une approche intéressante. A notre connaissance, il n'existe pas dans la littérature de travaux proposant l'utilisation de solveurs SAT pour la commande d'un système automatisé de production. Néanmoins, il existe des travaux utilisant de telles techniques pour traiter d'autres problèmes liés aux SED.

Voronov et Akesson (2008) proposent une approche à base de solveur SAT permettant la synthèse d'un superviseur au sens de la théorie de la commande par supervision (Wonham, 2015). Grastien et Anbulagan (2013) proposent d'utiliser des techniques SAT pour le diagnostic des SED. Ces approches sont basées sur la traduction des automates à états finis dans le formalisme SAT, ainsi que sur la traduction des propriétés à vérifier (contrôlabilité, diagnosticabilité, etc.).

L'utilisation de telles techniques semble donc possible pour le travail sur les systèmes à événements discrets, et plus particulièrement avec les approches de commande par contraintes (Hietter, 2009; Riera *et al.*, 2015).

I-5 Conclusion

Ce chapitre introductif a permis de présenter différents aspects liés à la conception d'un programme API.

Tout d'abord, la section I-1 a montré que la 4^{e} révolution industrielle et les préceptes définissant l'industrie du futur impliquent une évolution des techniques de conception d'un contrôleur. Le principe générique de la conception d'un contrôleur a également été introduit dans cette section.

Par la suite, des méthodes et outils disponibles dans l'industrie (section I-2) et dans le monde académique (section I-3) ont été présentés. Deux constats peuvent être soulevés :

— les méthodes de conception du monde industriel permettent difficilement d'atteindre les objectifs de l'industrie du futur en terme de flexibilité et de fiabilité des programmes API ;
— les méthodes formelles académiques de conception ne sont ni connues ni utilisées par l'industrie car trop complexes à appliquer sur un système industriel complet.

Enfin, la section I-4 a présenté des approches académiques alternatives ayant pour objectif de faciliter l'utilisation pratique d'approches formelles pour la conception d'un programme API. Dans ce but, différents auteurs ont proposé des approches à base d'équations logiques. Ce formalisme d'équations logiques à base de variables logiques, en comparaison aux approches à base de langages formels ou de réseaux de Petri, est plus proche des pratiques des automaticiens. La prise en main et l'utilisation de ces approches par le monde industriel est donc simplifié.

Dans cette thèse nous avons choisi de contribuer à une approche à base d'équations logiques appelée approche par filtre logique. Cette approche a l'avantage de pouvoir être utilisée de différentes manières en fonction des besoins : commander un système contrôlé par un API, vérifier formellement un programme API ou bien mettre en sécurité un programme API déjà existant présentant des erreurs. Néanmoins, suite aux derniers travaux sur le filtre logique (Riera *et al.*, 2015), différents verrous et points d'améliorations ont été identifiés.

Dans le but de proposer une méthode formelle, appuyée par des outils formels, s'intégrant dans les méthodologies industrielles, nous proposons de contribuer à l'approche par filtre logique à différents niveaux.

- La formalisation des contraintes logiques définissant le filtre logique est retravaillée pour améliorer l'expressivité de ces contraintes.
- La notion de priorité entre les variables d'une contrainte, présentée dans des travaux précédents (Coupat, 2014), est formalisée.
- La propriété de cohérence d'un filtre logique, introduite dans des travaux précédents (Riera *et al.*, 2015), est formalisée. De plus, une solution fournissant une condition nécessaire et suffisante pour la vérification formelle de cette propriété est proposée.
- Dans la méthodologie de base, la définition des contraintes logiques est effectuée manuellement à partir des exigences textuelles du cahier des charges. Dans cette thèse, une approche à base d'automates à états finis est proposée pour générer automatiquement les contraintes logiques.
- L'algorithme d'implémentation « classique » du filtre logique (Coupat, 2014), est mis à jour pour prendre en compte le nouveau formalisme.
- Deux algorithmes de recherche locale de solutions sont proposées en se basant sur des techniques de solveur SAT. Ces solutions d'implémentation permettent de simplifier les étapes de conception en amont de l'implémentation.

Le chapitre suivant a pour objectif d'illustrer les constats effectués au début de cette conclusion. Pour ce faire, deux expériences distinctes sont présentées et discutées. Par la suite, les contributions méthodologiques et théoriques sont présentées dans le chapitre III. Puis, le chapitre IV présente les contributions algorithmiques liées à l'implémentation du filtre logique dans un API. Enfin, le chapitre V applique l'approche par filtre logique sur différentes stations d'une cellule flexible réelle.

CHAPITRE II

Du cahier des charges à l'implémentation : exemples et motivations

II-1 Introduction

Le chapitre précédent a présenté différents points liés à la conception d'un programme API. Il apparait que deux grands principes existent : d'un côté le monde industriel cherche à obtenir une solution rapidement puis à itérer sur cette solution afin de l'améliorer, de l'autre côté le monde académique cherche à garantir la solution en utilisant des approches formelles.

Le constat est qu'il existe un écart méthodologique important entre ces deux mondes. En effet, les méthodes formelles peuvent être complexes à utiliser et peuvent impliquer un temps de conception plus long par rapport à une méthode de conception cyclique. Néanmoins, l'utilisation d'une méthode cyclique non-formelle ne permet pas de garantir à 100% que le résultat est correct par rapport aux exigences et critères.

L'objectif de ce chapitre est d'illustrer les différents points de ce constat, pour ce faire deux études ont été menées. Dans un premier temps, la section II-2 met en évidence un manque de méthodologie, que l'on soit étudiant ou expert du domaine. Dans un second temps, la section II-3 permet de montrer au travers d'un exemple simple, que l'utilisation des méthodes formelles « classiques » est très difficilement applicable à un système industriel réel.

II-2 Retour d'expérience sur la conception d'un contrôleur

Cette section présente une expérience, menée auprès d'étudiants ingénieurs et de docteurs, dans le domaine des sciences de l'ingénieur. Le but de cette étude est de confronter les participants à un problème simple de commande, puis d'observer et analyser les méthodes mises en place pour la réalisation du programme de commande. Dans un premier temps, le cas d'étude est présenté (section II-2.1). Puis, le protocole expérimental est décrit (section II-2.2). Enfin, une synthèse et une analyse des résultats sont proposées (section II-2.3).

II-2.1 Présentation du cas d'étude

L'objectif pour les participants est de concevoir un programme de contrôle/commande permettant la gestion automatique d'un portail. Le portail à commander est celui du logiciel de simulation HOME I/O [1] (Fig. 21), fruit d'une collaboration entre le laboratoire CReSTIC et la société RealGames (Riera *et al.*, 2016)

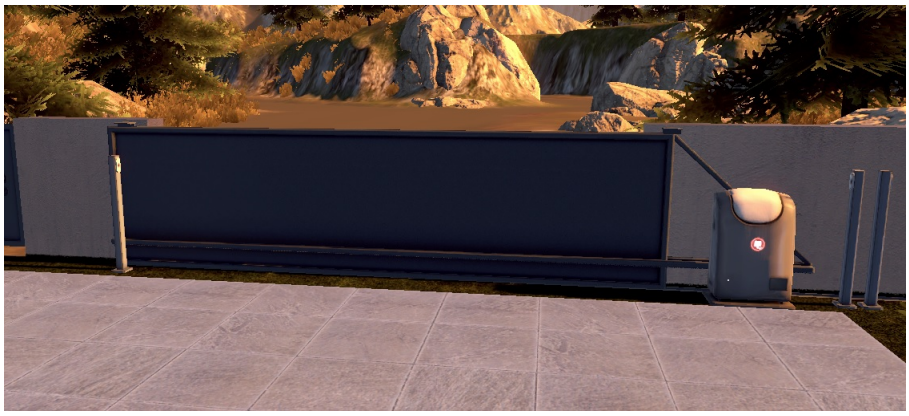


FIGURE 21 – Portail de HOME I/O

Pour commander les systèmes de HOME I/O (volets, chauffage, portail, éclairage, alarme...) plusieurs solutions sont possibles. Un mode de programmation interne à HOME I/O est proposé, ce mode permet la déclaration simplifiée de scénario pour l'initiation à la programmation. Un mode de programmation externe est également proposé, dans ce mode de fonctionnement la commande doit être effectuée en dehors de HOME I/O. Ce

1. `https://realgames.co/home-io/`

mode externe permet par exemple de se connecter à un automate programmable (simulé ou non), ou bien à d'autres types de contrôleurs (Arduino, Raspberry Pi...) via différents protocoles de communication.

Pour le bon déroulement de l'expérimentation, nous souhaitions que les participants développent leurs programmes dans un environnement non conventionnel qu'ils ne connaissaient pas avant. Dans ces conditions, d'une part la comparaison des résultats n'est pas biaisée par la connaissance préalable des outils par les participants. D'autre part, la séparation entre les étapes de spécification et d'implémentation est bien présente. Nous avons choisi d'utiliser l'environnement Scratch 2 pour la programmation, car celui-ci est facile à prendre en main (il est utilisé en collège) et ne nécessite donc pas de formation préalable. HOME I/O propose de base une communication [2] avec le logiciel Scratch 2.

Scratch 2 [3] est un logiciel développé par le MIT (Institut de Technologie du Massachussets), il permet l'apprentissage ludique de la programmation. La programmation se fait à l'aide de blocs s'imbriquant les uns les autres, différents blocs existent allant de l'affectation de variables à l'émission de messages en broadcast à l'ensemble des blocs du programme. Il est assez intuitif de développer un algorithme basique utilisant les notions de base de programmation (tests conditionnels, boucles, calculs...). Scratch permet également le développement de compétences en programmation plus évoluées telles que la programmation événementielle, l'animation d'objets graphiques et sonores ou bien encore la programmation parallèle.

L'interface de Scratch 2 (Fig. 22) est découpée en trois parties principales. La partie de gauche permet l'affichage des variables et des animations. La partie centrale contient l'ensemble des blocs utilisables. Enfin, la partie de droite est l'espace où l'on assemble les blocs pour réaliser le programme.

Pour finir sur la présentation du cas d'étude, les éléments liés au portail à commander ainsi que le fonctionnement attendu du programme sont détaillés.

Un moteur permet de mettre en mouvement le portail. Trois actions sont possibles sur ce moteur : *ouvrir*, *fermer*, *stopper*. Deux capteurs tout ou rien de fin de course sont instrumentés sur le système (*fermé*, *ouvert*). Ceux-ci permettent de savoir si le portail est

2. `https://realgames.co/docs/homeio/fr/scratch2/`
3. `https://scratch.mit.edu`

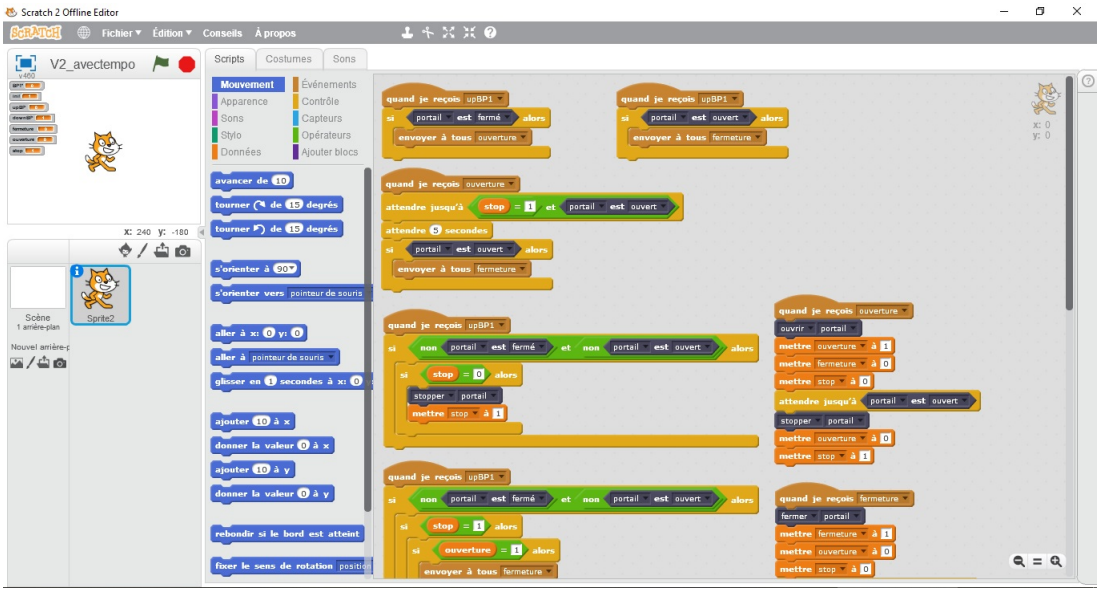


FIGURE 22 – Interface de Scratch 2

totalement ouvert ou totalement fermé. Enfin, une télécommande est disponible. Dans cette expérimentation, seul le bouton 1 de la télécommande est utilisé pour commander entièrement le système (noté *bp1*).

Le cahier des charges du contrôleur est décrit ci-dessous :

- Lorsque le portail est fermé :
  un appui sur bp1 ouvre le portail
- Lorsque le portail est en train de s'ouvrir :
  un appui sur bp1 stoppe le portail
  un nouvel appui sur bp1 ferme le portail
- Une fois ouvert, le portail se referme automatiquement après 5 secondes :
  Si on appuie sur bp1 quand le portail est ouvert, celui-ci se referme immédiatement
- Lorsque le portail est en train de se fermer :
  un appui sur le bp1 stoppe le portail
  un nouvel appui sur bp1 ouvre le portail
- Le moteur doit s'arrêter lorsque le portail est ouvert ou fermé
- Il n'est pas demandé de gérer la détection de personnes et/ou de véhicules.

Il est important de souligner que ce cahier des charges fonctionnel est incomplet volontairement. En effet, l'état initial du système n'est pas précisé. De plus, le terme « appui » sur

un bouton est ambigu : front montant, front descendant ou bien appui long durant un certain temps... Cela permettra d'observer comment les participants traitent ces incertitudes et comment proposent-ils d'y répondre.

Après cette description générale de l'étude ainsi que des outils logiciels utilisés, la section suivante présente le protocole expérimental (section II-2.2).

II-2.2 Protocole expérimental

Comme introduit précédemment, cette étude a été proposée à deux groupes de participants différents. Le groupe A est composé d'étudiants en deuxième année d'école d'ingénieur (BAC+4). Ces étudiants suivent un cursus spécialisé en automatique et informatique industriel. Le groupe B est composé d'enseignants chercheurs et de doctorants travaillant également dans les domaines de l'automatique et de l'informatique industrielle.

Le même problème a été posé aux deux groupes, néanmoins le suivi de l'avancement des groupes n'a pas été le même. Cette section présente les spécificités de chacun des groupes.

II-2.2.1 Groupe A

Le groupe A est composé de 19 étudiants de deuxième année d'école d'ingénieur, spécialisés en automatique et informatique industrielle. Les étudiants ont travaillé en binôme et un étudiant a travaillé seul. Par soucis d'anonymat, les binômes sont nommés **B1** à **B9** et l'étudiant seul **S1**. Pour ce groupe A, les participants ont été observés pendant 4h de travaux pratiques réparties sur deux séances :

- Séance 1 : 2h vendredi matin (jour J)
  - Découverte du problème (sujet disponible en Annexe C)
  - Découverte des logiciels (démonstration rapide par les encadrants)
  - Observations et réponses aux questions par les encadrants
- Séance 2 : 2h lundi soir (jour J+3)
  - Observations et réponses aux questions par les encadrants
- Date de rendu du livrable : jour J+10

Entre les séances, les étudiants avaient la possibilité de travailler chez eux, la salle d'enseignement était également à leur disposition. Il est également à noter que tous les étudiants connaissaient déjà le logiciel HOME I/O et que 2 binômes (B5 et B8) connaissaient Scratch 2 mais n'avaient jamais programmé avec.

II-2.2.2 Groupe B

Le groupe B est composé de 6 participants, tous ces participants sont docteurs ou doctorants en automatique. Par soucis d'anonymat, ces participants sont nommés **D1** à **D6**.

Étant donné que ce groupe est composé d'experts du domaine, une description et un accompagnement minimal ont été nécessaires. En effet, seuls le fonctionnement attendu présenté précédemment (section II-2.1), les liens de téléchargements des logiciels ainsi que les explications pour connecter HOME I/O et Scratch 2 ont été fournis.

Pour ce groupe, aucune contrainte temporelle n'a été donnée. De plus, pour avoir un retour méthodologique, il a été demandé aux participants du groupe B d'expliquer la démarche utilisée et/ou le principe de programmation utilisé.

II-2.3 Synthèses et analyses des résultats

II-2.3.1 Synthèse des résultats du groupe A

Pour présenter les résultats, les points suivants de la conception d'un programme sont détaillés : l'analyse, la conception, la vérification puis la validation par les clients (ici les encadrants). L'étape d'implémentation n'est pas détaillée car celle-ci n'est pas nécessaire avec l'utilisation de Scratch 2. En effet, le protocole de communication permet l'échange des informations (entrées, sorties), directement entre le contrôleur (Scratch 2) et le système à contrôler (HOME I/O).

Analyse du problème et modélisation Après une lecture très rapide (voire incomplète) du sujet, la plupart des groupes ouvrent les logiciels et commencent à programmer. Seuls les groupes B1, B3 et B6 prennent une feuille et commencent à spécifier le programme à l'aide d'un Grafcet. Après environ 30 minutes, seuls les groupes B7 et S1 n'ont pas utilisé

de méthode de spécification de leur programme, ces groupes n'en feront d'ailleurs jamais pendant l'expérimentation. Le tableau 3 indique la méthode utilisée par les participants.

Table 3 – Approche de spécification utilisée

	B1	B2	B3	B4	B5	B6	B7	B8	B9	S1
Grafcet	X	X	X	X		X		X		
Organigramme					X					
Algorithme									X	

Au final, la majorité des groupes ont commencé à structurer leur pensée en utilisant une feuille et un stylo. Néanmoins, il est apparu assez rapidement que ces Grafcet, algorithmes et organigrammes sont restés incomplets et ont été mis de côté après 1h de travail pour presque tous les groupes. En effet, seuls B1, B2 et B3 ont continué d'utiliser leurs Grafcet et ont proposé une approche d'implémentation permettant la traduction d'un Grafcet dans Scratch.

En moyenne, les groupes déclarent dans leurs livrables avoir passer 46 minutes pour l'étape d'analyse du problème (lecture du sujet, prise en main des logiciels et modélisation).

Conception : principes et architectures Initialement, tous les groupes commencent avec une architecture de programmation que l'on peut qualifier de « centralisée » : une seule boucle infinie qui est activée à l'initialisation du programme et qui englobe l'ensemble des tests et actions. Au fur et à mesure de l'avancement des groupes, d'autres architectures de programme apparaissent. Certains utilisent une architecture « parallèle », c'est-à-dire que différents blocs indépendants sont actifs en même temps. D'autres utilisent une architecture « décentralisée » : un seul bloc est actif en même temps, mais plusieurs blocs indépendants existent. Le tableau 4 présente les architectures finales utilisées par les groupes.

Table 4 – Architectures utilisées

	B1	B2	B3	B4	B5	B6	B7	B8	B9	S1
Centralisée					X	X	X		X	X
Parallèle		X	X					X		
Décentralisée	X			X						

En plus des architectures utilisées, deux points intéressants sont à souligner : la gestion de l'état initial et la gestion des appuis longs sur le bouton.

Seuls les groupes B2 et B8 ont intégré à leur programme une gestion robuste de l'état initial. C'est-à-dire que, quel que soit l'état initial du portail, le programme doit être capable de fonctionner.
Bien que tous les groupes aient posé des questions sur la gestion de l'appui prolongé sur le bouton, seuls les groupes B3, B5, B6, B8 et B9 ont proposé une approche pour capturer un front (montant ou descendant) du bouton et de l'utiliser pour déclencher les actions. Pour réussir à « capturer » ces fronts, les groupes B3, B5, B6 et B8 ont utilisé des temporisations, mais ces temporisations insèrent un délai gênant dans l'utilisation du portail. Finalement, seul le groupe B9 a proposé une structure permettant de capturer un front sans insérer de temporisation dans le programme.

Méthodes de tests de vérification En ce qui concerne les tests, tous les participants de ce groupe ont travaillé de la même façon. Concrètement, dès qu'un bloc est ajouté ou une condition logique est modifiée, les participants mettent en fonctionnement le programme et vérifie que le système contrôlé « réagit bien ». Les participants réalisent donc en permanence des « essais-erreurs » sur le système puis modifient leur programme et recommencent leurs tests.

Cette approche de test est possible car le système à contrôler est simulé (virtual commissioning), une erreur n'implique donc pas de dégradation ou de danger. Cependant, cette approche de correction d'une erreur implique assez souvent la génération d'une autre erreur (nouvelle ou corrigée précédemment). Les participants effectuent donc assez souvent des allers-retours entre les tests et la correction du programme.

Tous les participants ont indiqué que le problème était très simple et donc ne nécessitait pas de vérification formelle. De plus, ils ont tous conclu dans leur livrable que leur programme était correct (c'est-à-dire répond entièrement au cahier des charges et sûr de fonctionnement).

Validation par les encadrants Afin de valider les programmes, les encadrants ont appliqué une méthode à base de « cahier de recettes ». Le principe est d'établir un ensemble de scénarios (principe d'action/réaction). Chaque scénario doit permettre de détecter des dysfonctionnements. Le tableau 5 présente les scénarios utilisés.

TABLE 5 – Exemple de scénarios de validation

n°	Conditions	Actions	Résultats attendus
1	portail fermé	appui sur BP1	ouverture du portail
2	portail ouvert	appui sur BP1	fermeture du portail
3	portail en ouverture	appui sur BP1	arrêt du portail
4	portail en fermeture	appui sur BP1	arrêt du portail
5	après n° 3	appui sur BP1	fermeture du portail
6	après n° 4	appui sur BP1	ouverture du portail
7	portail fermé		moteur arrêté
8	portail ouvert		moteur arrêté
9	après ouverture complète	attendre 5 secondes	fermeture
10	juste après n° 9	faire n° 4 puis n° 9	le portail doit se refermer après 5 sec (timer bien réinitialisé)
11	quelle que soit l'étape	appui maintenu sur BP1	même fonctionnement qu'avec un appui « simple »
12	portail fermé, semi-ouvert ou ouvert	initialisation du programme	fonctionnement normal (pas de deadlock)

Tous les programmes ont été soumis aux mêmes scénarios. Bien que tous les groupes aient été sensibilisés pendant la séance aux problèmes de sûreté de fonctionnement, très peu d'entre eux ont réussi à détecter des erreurs dans leurs programmes. De plus, une fois les erreurs détectées (par les encadrants), en fin de seconde séance en général, ils n'ont pas pris le temps de corriger ces erreurs qui entraînaient très souvent la reprise totale de leur architecture de programme.

Le tableau 6 synthétise quelques points abordés précédemment. Nous pouvons remarquer que presque tous les participants ont spécifié leur programme (Grafcet, algorithme ou organigramme), néanmoins aucun n'a senti le besoin de vérifier formellement ou exhaustivement leurs solutions. Seuls B6, B8 et B9 répondent au cahier des charges de base. De plus, B8 est le seul à répondre au cahier des charges tout en gérant les problèmes de sûreté de fonctionnement non indiqués explicitement dans le cahier des charges. Enfin, les groupes B7 et S1 ont été les seuls à ne pas proposer de spécification.

TABLE 6 – Résultats synthétisés du groupe A

	B1	B2	B3	B4	B5	B6	B7	B8	B9	S1
Spécification/modélisation	X	X	X	X	X	X		X	X	
Vérification formelle										
Répond au cahier des charges						X		X	X	
Gestion des appuis prolongés								X		
Robuste à l'état initial								X		

II-2.3.2 Synthèse des résultats du groupe B

Les participants du groupe B n'ont pas été observés, par conséquent seuls le principe des programmes et les explications fournies par certains participants sont présentés ici. Pour faciliter la comparaison des groupes A et B nous proposons de suivre la même présentation des résultats que précédemment.

Analyse du problème et modélisation D1, D2, D3 et D6 ont utilisé une approche de spécification du problème (Tab. 7). D5 indique n'avoir utilisé aucun moyen de spécification et D4 ne donne pas d'information à ce sujet.

TABLE 7 – Approche de spécification utilisée

	D1	D2	D3	D4	D5	D6
Grafcet	X					
Automate à états		X	X			X

Conception : principes et architectures En comparant au groupe A, le groupe B propose des architectures de programme plus structurées et réfléchies.

D1 propose une traduction de son grafcet en utilisant des blocs séparés et la notion de message. A chaque bloc va correspondre une étape du grafcet. La transition d'une étape à l'autre va se faire par l'échange de message (proche de la notion d'événements). D1 indique également avoir tenté une approche centralisée, avant de proposer cette traduction de grafcet.

D2 indique avoir utilisé un automate à états pour structurer sa pensée et son programme. Néanmoins, l'architecture centralisée de son programme est difficilement compréhensible.

D3 propose une implémentation de son automate à états en utilisant une architecture centralisée. Une boucle est active, correspondant à un état, tant que la condition de sortie de cet état n'est pas validée. De plus, tant que l'état est actif, une action peut être effectuée dans la boucle.

D4 utilise une architecture décentralisée. Néanmoins, aucun principe de programmation ou d'organisation du programme n'est indiqué par le participant.

D5 propose de reconstruire les états du portail (ouvert, fermé, ouverture, fermeture) ainsi que les événements associés aux capteurs (fronts montant et descendant) ainsi qu'aux actions (ouverture, fermeture). L'idée de la modélisation de ce participant est de se rapprocher du fonctionnement des systèmes à événements discrets tels que décrits dans la littérature. L'architecture proposée est parallèle, plusieurs blocs indépendants fonctionnent simultanément. Chaque bloc permet de réagir à un événement (en utilisant la notion de message de Scratch).

D6 propose également la modélisation des états du système et des événements associés. L'architecture est parallèle, les blocs sont tous actifs en même temps, les informations liées aux états et aux événements sont stockées dans des variables partagées.

Le tableau 8 présente les mêmes critères de comparaison que pour le groupe A.

Table 8 – Architectures utilisées

	D1	D2	D3	D4	D5	D6
Centralisée		X	X			
Parallèle					X	X
Décentralisée	X			X		

Validation Pour la validation, les mêmes scénarios ont été utilisés pour les groupes A et B (Tab. 5).

Table 9 – Résultats synthétisés du groupe B

	D1	D2	D3	D4	D5	D6
Spécification/modélisation	X	X	X			X
Vérification formelle						
Répond au cahier des charges	X	X	X	X	X	X
Gestion des appuis prolongés		X	X	X	X	X
Robuste à l'état initial				X	X	X

Dans ce groupe B, l'ensemble des participants ont réussi à répondre au cahier des charges. Nous souhaitons préciser que D1 a introduit des temporisations qui réduisent la réactivité du programme, néanmoins d'un point de vue fonctionnel le programme est bon. D2 et D3 ont proposé une solution pour gérer les différents états initiaux possibles, néanmoins ils ont tous les deux un problème si le portail est totalement ouvert à l'initialisation.

II-2.4 Analyse des résultats

Cette expérience et les résultats présentés précédemment, permettent de mettre en évidence différents problèmes méthodologiques. Cette section présente une analyse globale des résultats des deux groupes de participants.

Que l'on soit étudiant ou expert du domaine, il apparaît clairement que face à un problème concret, l'étape de spécification du cahier des charges n'est pas suffisamment traitée. La simplicité du problème posé ici peut expliquer un tel comportement, néanmoins cela s'observe assez souvent dans l'industrie où l'on va préférer développer rapidement un prototype plutôt que de « perdre » du temps sur une étape de formalisation.

Un deuxième point à soulever est l'absence de vérification formelle pour l'ensemble des participants. L'outil de développement Scratch ne permet pas une vérification formelle ou exhaustive du programme. Pour les participants n'ayant pas spécifié/formalisé leur solution, la seule possibilité était donc de traduire (manuellement) le programme Scratch dans un autre outil permettant une analyse. Néanmoins, ceux ayant utilisé un Grafcet ou un automate à états finis n'ont également pas vérifié formellement leur solution. Deux raisons à cela ont été soulevées avec les participants, la première est qu'ils n'ont pas ressenti la nécessité car le problème semblait simple, la seconde est qu'ils n'avaient pas connaissance d'outils permettant la vérification (notamment lors d'utilisation de Grafcet).

Tous les participants ont exprimé le fait que le problème était « trop simple » pour nécessité une formalisation, ou une vérification formelle. Ils n'ont donc pas souhaité prendre le temps nécessaire à une bonne modélisation et/ou vérification. Néanmoins, peu d'entre eux ont réussi à répondre au cahier des charges et encore moins en garantissant la sûreté de fonctionnement ou la robustesse du programme.

En analysant les programmes finaux, il est assez intéressant de remarquer qu'ils sont tous

différents. Même lorsque le point de départ est très proche, par exemple tous les Grafcet des participants sont assez proches les uns des autres, néanmoins, les implémentations de ces Grafcets sont tous différents.

En ce qui concerne l'outil Scratch, les participants du groupe A (étudiants) n'ont eu aucune difficulté à l'utiliser. Durant les séances d'observation, aucun participant du groupe A n'a souhaité avoir d'information sur le fonctionnement de Scratch et son interfaçage avec le logiciel de simulation HOME I/O. Les participants du groupe B (experts du domaine), ont en général éprouvé le besoin de comprendre plus précisément le fonctionnement de Scratch, ainsi que de sa connexion à la partie opérative simulée (HOME I/O), avant de pouvoir proposer un programme de commande.

D'un point de vue général, cette expérience montre que les méthodologies employées, pour l'obtention d'un programme de commande, sont dépendantes des expériences et des affinités de chacun. De plus, à un même problème initial, avec les mêmes contraintes techniques, de nombreux résultats différents sont possibles. Ces résultats répondent à tout ou partie du cahier des charges, mais il est difficile de vérifier et de valider l'exactitude et la complétude de ces résultats.

Afin de pouvoir plus facilement garantir le résultat, mais également de réduire le biais lié à l'utilisateur d'une méthode, le monde académique propose des méthodes formelles. La section II-2.5 présente des travaux appliquant la théorie de la commande par supervision (SCT) sur l'exemple du portail.

II-2.5 Application de la SCT à l'exemple du portail

Cette section a pour objectif de montrer comment la SCT, sous certaines hypothèses (asynchronisme, déterminisme), permet de concevoir un programme de commande implémentable dans un Automate Programmable Industriel (API). L'approche et les résultats présentés ici sont extraits de (Roussel et Giua, 2005; Cantarelli et Roussel, 2008).

Pour rappel, le principe de la SCT est de modéliser, à l'aide d'automates à états finis, le système à contrôler ainsi que les exigences. A partir de ces modèles, un superviseur est calculé automatiquement (section I-3.2). Néanmoins, l'utilisation de la SCT dans le cas particulier de la programmation d'API, pose de nombreux problèmes (section I-3.3).

Les travaux résumés ici (Roussel et Giua, 2005; Cantarelli et Roussel, 2008), ont pour objectif de proposer une méthodologie globale permettant de résoudre certains de ces problèmes. L'idée principale est la prise en compte, dès la modélisation, du fonctionnement final de l'API. Ce faisant, le passage du superviseur (résultat de la SCT) au programme API, est facilité. La méthodologie globale est résumé ci-après :

- modélisation générique des capteurs et des actionneurs ;
- modélisation générique du fonctionnement cyclique de l'API ;
- modélisation du fonctionnement de la partie opérative ;
- modélisation des spécifications ;
- calcul et réduction du superviseur ;
- extraction du contrôleur à partir du superviseur ;
- transformation du contrôleur en machine de Mealy ;
- implémentation de la machine de Mealy dans un langage API.

L'application de cette méthodologie, à l'exemple du portail résulte en différents modèles présentés dans Cantarelli et Roussel (2008). Il est à noter que le cahier des charges est légèrement différent de celui utilisé précédemment (section II-2.1) : un capteur de présence est utilisé pour la détection d'un véhicule, et aucune temporisation n'est demandée entre l'ouverture complète et la fermeture automatique. Les résultats quantitatifs, en terme de nombre de modèles et de leurs tailles, sont présentés dans le tableau 10.

Table 10 – Résultats quantitatifs de l'approche (Cantarelli et Roussel, 2008)

	automates	états	transitions
Modèles du système	11 (4 pour les entrées, 2 pour les sorties, 1 pour l'API, 4 pour la partie opérative)	481	1330
Spécifications	11	276	1215
Superviseur	1	368	646
Sup. réduit	1	110	194
Contrôleur	1	45	110
Mealy	1	15	40

Ces travaux montrent qu'il est possible d'utiliser la SCT pour la conception d'un programme API. Néanmoins, différentes limitations peuvent être soulevées.

Premièrement, le résultat final est très dépendant de l'état initial choisi. Par conséquent, le programme API, et donc la méthodologie complète, doit être ré-appliquée dès que l'état initial du système doit être modifié. Or, dans le cas de l'industrie du futur, les équipements

doivent être de plus en plus réutilisables et flexibles, cela implique que les programmes doivent être robustes à des états initiaux différents.

Deuxièmement, la modélisation par automates à états finis est soumise à l'hypothèse d'asynchronisme des événements. Or, dans un API, différents événements peuvent apparaître simultanément à cause du temps de cycle de l'API. Les auteurs proposent une modélisation MIMO (multiple input multiple output) de l'API plutôt qu'une modélisation SISO (single input single output). Néanmoins, la prise en compte d'événements simultanés (ou synchronisés) n'est pas discutée dans ces travaux.

Enfin, malgré la simplicité de l'exemple utilisé (portail automatique), les calculs intermédiaires (superviseur) impliquent des modèles comportant des centaines d'états. Le passage à l'échelle, sur un système industriel complexe, impliquera donc forcement un problème d'explosion combinatoire.

La suite de ce chapitre (section II-3), met en évidence les problèmes liés aux hypothèses classiques de la SCT (déterminisme et asynchronisme). De plus, une approche de modélisation est proposée. Celle-ci permet d'apporter des éléments de réponses vis à vis de la modélisation des événements synchronisés, ainsi que de la gestion robuste des états initiaux possibles.

II-3 Nécessité d'adaptation des approches formelles

Le chapitre précédent (section I-3.3) a montré que l'utilisation de la Théorie de la Commande par Supervision (SCT), pour la synthèse d'un superviseur à implémenter dans un API (contrôleur), soulève de nombreux problèmes. Cette partie a pour objectif de montrer comment il est possible, à l'aide de principes de modélisation, de « contourner » certaines hypothèses fondamentales de la SCT (asynchronisme et déterminisme), afin de faciliter l'implémentation du superviseur dans un Automate Programmable Industriel (API).

Dans cette section, un exemple bien connu de la littérature est utilisé pour illustrer l'approche : le chat et la souris. De plus, le logiciel Supremica (Akesson *et al.*, 2006) est utilisé pour la modélisation et les calculs.

Dans Ramadge et Wonham (1989), les auteurs proposent une maison composée de 5

pièces (Fig. 23). Dans cette maison sont placés un chat et une souris. Ceux-ci peuvent se déplacer dans les différentes pièces en empruntant des portes qui leurs sont dédiées : les portes $\{C1, \ldots, C6, c7\}$ sont utilisables par le chat, les portes $\{M1, \ldots, M6\}$ sont utilisables par la souris.

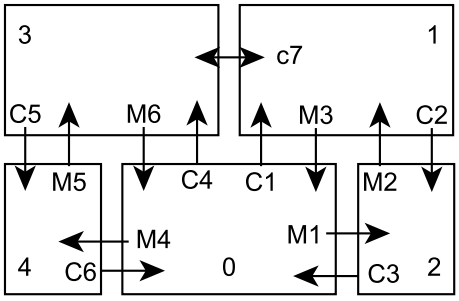


FIGURE 23 – Représentation de la maison

Les portes ne peuvent être franchies que dans un seul sens (indiqué par une flèche sur la Fig. 23), seule la porte $c7$ est franchissable dans les deux sens. Par ailleurs, toutes les portes sont commandables en ouverture/fermeture à l'exception de $c7$ qui est tout le temps ouvert ($c7$ s'apparente donc à un tunnel).

Dans le problème classique décrit dans Ramadge et Wonham (1989), le chat est initialisé dans la pièce 2 et la souris dans la pièce 4. Le but est alors de proposer une loi de commande automatique des portes permettant de garantir que, quel que soit le mouvement des animaux, le chat ne pourra jamais se trouver dans la même pièce que la souris.

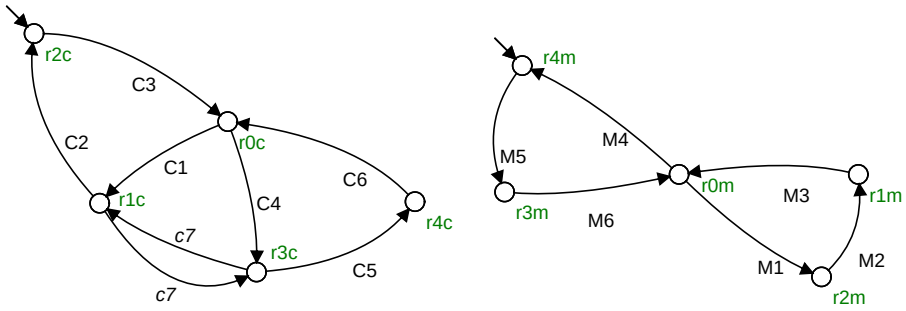
Dans la section II-3.1 est présentée une modélisation permettant l'obtention du résultat classique de la SCT (Ramadge et Wonham, 1989). La section II-3.2 met en évidence deux problématiques : l'hypothèse d'asynchronisme des événements et l'hypothèse de déterminisme entrainant un état initial unique. La section II-3.3 propose deux ajouts à la modélisation permettant de traiter ces problématiques.

II-3.1 Le chat et la souris : résultats classiques

II-3.1.1 Modélisation du système

En se basant sur l'organisation de la maison présentée figure 23, les modèles des mouvements du chat et de la souris sont définis avec deux automates à états finis (Fig. 24).

Chaque état représente une position, c'est-à-dire une pièce, et chaque arc représente un déplacement possible en traversant une porte.



(a) Chat (b) Souris

FIGURE 24 – Modèles des mouvements

Le modèle global du système G est calculé par composition parallèle : $G = Chat \| Souris$. G contient 25 états, 13 événements et 70 transitions, une représentation matricielle de cet automate est proposée dans Ramadge et Wonham (1989). Ce modèle G contient tous les mouvements possibles des animaux, incluant la possibilité pour chacun d'eux d'être dans la même pièce simultanément.

Néanmoins, ces situations de proximités sont dangereuses pour la survie de la souris. Par conséquent, il est nécessaire de pouvoir calculer un superviseur contrôlant l'ouverture des portes, dans le but d'éviter au chat et à la souris de se retrouver dans la même pièce. Pour faire cela, cet objectif doit être défini formellement. Ceci peut être fait en ajoutant des automates à états finis appelés automates de *spécification*.

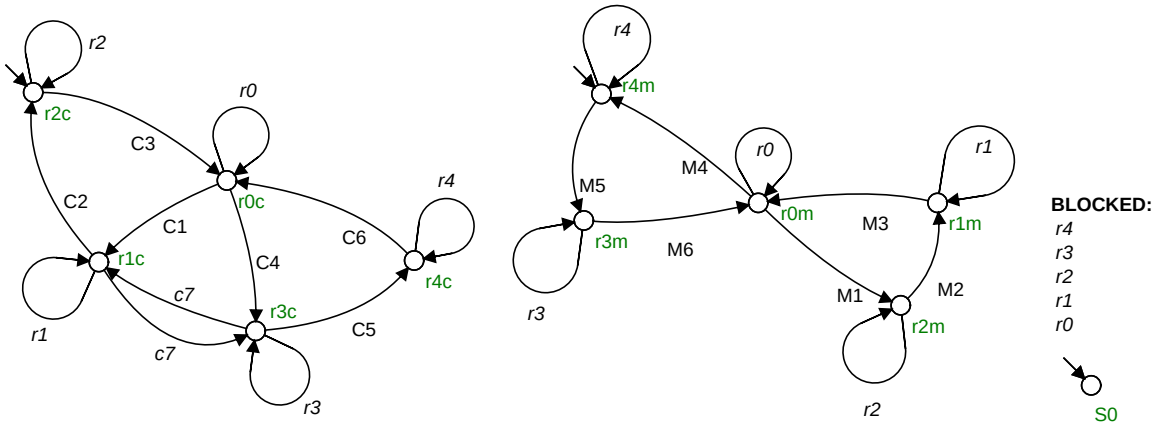
II-3.1.2 Modélisation des exigences : spécification

L'objectif principal est de synthétiser le superviseur le plus permissif possible qui empêche la possibilité pour le chat d'atteindre la souris (et vice versa). Il existe déjà plusieurs façons de modéliser cette interdiction (Ramadge et Wonham, 1989; Miremadi *et al.*, 2008), nous proposons ici d'appliquer une approche modulaire à base d'états interdits.

L'approche utilisée est présentée dans Magnusson *et al.* (2010) et a pour but de modéliser facilement des états interdits. Le principe est d'ajouter aux modèles, un événement ev non-contrôlable, qui sera toujours interdit. Cet événement ev est ajouté sur un arc rebouclant (self-loop) sur l'état spécifique à interdire. Il est également possible avec cette approche

d'interdire à plusieurs états d'être actifs simultanément.

Lors de l'application des algorithmes de la SCT, les boucles non-contrôlables comportant l'événement *ev* survivront à la composition uniquement lorsque le chat et la souris seront dans la même pièce. Puis les spécifications interdiront cet événement, par conséquent, lors de l'étape de synthèse de la SCT, les états où le chat et la souris sont dans la même pièce seront supprimés du superviseur.



(a) Chat (b) Souris (c) Spécification

FIGURE 25 – Modèles des spécifications

La figure 25 détaille comment l'exigence « le chat et la souris ne doivent pas être dans la même pièce » est traduite en utilisant le principe des *listes d'événements bloqués* de Supremica. L'automate 25c contient la liste avec 5 événements non-contrôlables à interdire ($\{r0, \ldots, r4\}$ pour les pièces 0 à 4), les automates 25a et 25b ont des arcs rebouclant avec les événements correspondants.

Les événements r_i doivent être interdits, mais ces événements ne peuvent apparaître que si le chat et la souris sont dans la même pièce i. Par conséquent, avec cette modélisation, chaque couple $< ric, rim >$ est considéré comme interdit, ce qui est une formalisation correcte de l'exigence.

II-3.1.3 Synthèse du superviseur

Dans la formulation initiale du problème du chat et de la souris (Ramadge et Wonham, 1989), l'état initial est un état marqué. Cela signifie que l'état initial est un état objectif

qui doit être accessible depuis n'importe quelle configuration. Il est possible de considérer une variante à ce problème : tous les états sont marqués. En d'autres termes, chaque état peut être un état objectif car il n'y a pas de position spécifique à atteindre pour le chat et la souris.

Avec ces hypothèses, le superviseur résultant en considérant l'état initial $< r2c, r4m >$ est présenté dans la figure 26.

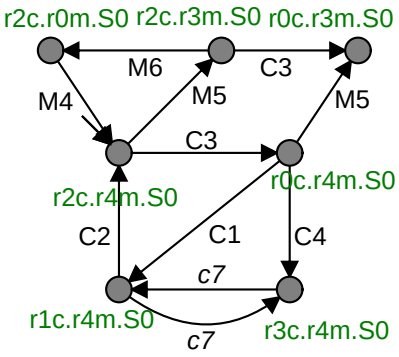


FIGURE 26 – Superviseur avec hypothèses classiques

II-3.2 Hypothèses classiques et limitations

La plupart du temps dans les systèmes à événements discrets, et en particulier dans la SCT, deux principales hypothèses sont retenues : le déterminisme et l'asynchronisme. L'asynchronisme signifie que deux événements différents ne peuvent pas apparaître simultanément. Le déterminisme signifie qu'il y a un unique état initial et que, pour chaque état, il n'existe pas deux arcs sortants de cet état portant le même événement.

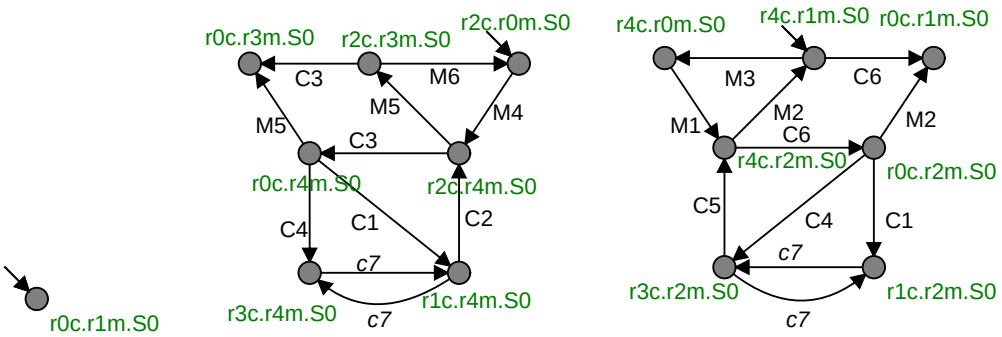
En considérant la définition de non-blocage d'un automate à états finis (Déf. 37), le superviseur de la figure 26 est non-bloquant puisque tous ses états sont marqués. Néanmoins, l'état $r0c.r3m.S0$ peut être qualifié de *deadlock* (Déf. 38), c'est-à-dire que, une fois dans cet état, le superviseur ne pourra plus évoluer vers aucun autre état et par conséquent le chat sera bloqué dans la pièce 0 et la souris dans la pièce 3 indéfiniment.

Le superviseur présenté dans la figure 26 est correct en considérant les hypothèses classiques et l'état initial choisi. Néanmoins, si l'état initial doit être changé, alors la méthodologie doit être à nouveau appliquée depuis le départ afin de synthétiser un nouveau

superviseur cohérent avec ce nouvel état initial. Différents superviseurs pour différents états initiaux sont présentés sur la figure 27.

Si l'on considère des cas d'applications réels ayant des exigences en terme de flexibilité, comme les systèmes manufacturiers par exemple, cet état initial fixé peut être un problème. En effet, la synthèse d'un superviseur pour un système de grande dimension demande un grand temps de calcul, donc il n'est pas possible de calculer un nouveau superviseur dès que les exigences changent (changement de pièces à produire, modification du flux de produit, etc.).

Une solution pourrait être de synthétiser hors-ligne un superviseur pour chaque état initial possible, puis d'utiliser en-ligne celui correspondant à l'état initial courant du système. Cependant, cela implique de générer un très grand nombre d'états et de transitions puis de les implémenter tous dans un contrôleur qui n'aura pas obligatoirement la capacité (mémoire, puissance de calcul) pour les traiter. De plus, cette méthode implique des questions de synchronisation et d'autorisation de changement de superviseur. Ces approches sont connues dans la littérature sous le terme de *reconfiguration* (Faraut *et al.*, 2010; Zhang *et al.*, 2015; Macktoobian et Wonham, 2017).



(a) r0c_r1m (b) r2c_r0m (c) r4c_r1m

FIGURE 27 – Superviseurs pour différents états initiaux

Si l'objectif est d'utiliser le superviseur dans un automate programmable réel, l'hypothèse d'asynchronisme peut amener à atteindre des états non-contrôlables voire dangereux. En effet, à cause du mécanisme cyclique d'un automate programmable, des événements différents peuvent apparaître simultanément. Si ces événements simultanés ne sont pas modélisés dès le début dans le superviseur, alors la décision finale prise par le superviseur aura de grandes chances d'être mauvaise.

Prenons par exemple le superviseur de la figure 27c, les animaux sont initialisés en $< r4c, r1m >$ et les événements (portes) disponibles sont $C6$ et $M3$. Mais ces deux portes permettent aux animaux d'atteindre la pièce 0. Donc, si le chat et la souris décident de se déplacer en même temps (et c'est possible car nous ne contrôlons pas leurs déplacements), ou si le temps de cycle est trop long et « capture » les événements dans un même cycle, alors le superviseur ne sera pas capable de protéger la souris du chat. L'hypothèse d'asynchronisme suivie, lors de la modélisation et de la synthèse, pose alors un problème vital pour la souris dès que l'on passe à une étape d'implémentation sur automate programmable.

La suite de cette section discute de ces problèmes : réduire la contrainte de l'état initial lié à l'hypothèse de déterminisme, et contourner l'hypothèse d'asynchronisme pour le contrôle des systèmes à événements discrets par API.

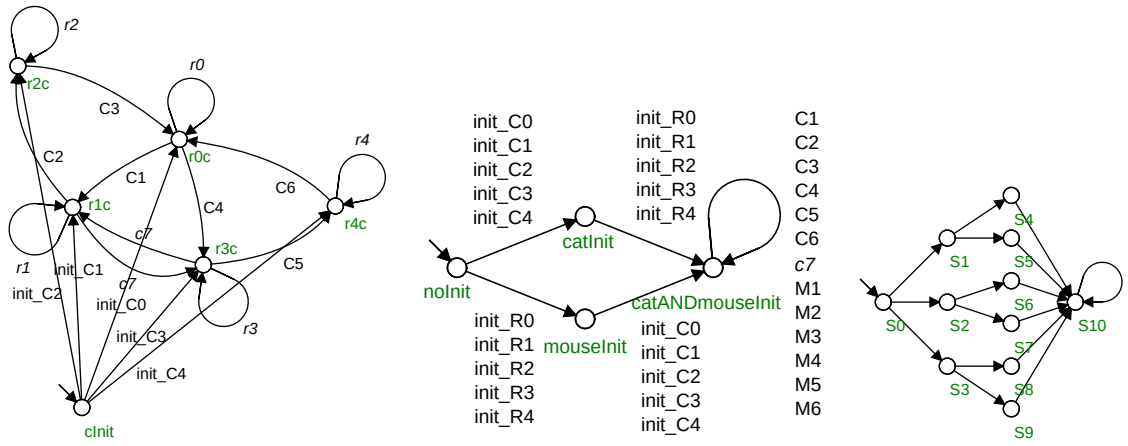
II-3.3 Hypothèses moins restrictives pour l'approche SCT

II-3.3.1 Prise en compte d'un état initial global

L'hypothèse de déterminisme est importante pour garantir le fonctionnement d'un système. Il est alors obligatoire de conserver l'existence d'un état initial. Néanmoins il est possible de réduire le conservatisme de l'état initial classique en synthétisant un unique superviseur cohérent avec tous les états initiaux. Pour ce faire nous proposons une approche de modélisation basée sur l'ajout d'un *état initial global* et d'un modèle d'initialisation.

L'état initial global est obtenu en ajoutant un état initial non marqué à chaque modèle du système (pas les spécifications). Puis, cet état est relié par des transitions à chaque état du modèle en question. Sur chacune de ces nouvelles transitions est associé un nouvel événement contrôlable représentant l'initialisation.

La figure 28a illustre le principe de modélisation pour le modèle du chat : un état initial est ajouté ($cInit$), puis une transition est ajoutée entre cet état et chacun des autres états initialisables du modèle (dans ce cas tous les états), enfin des événements sont créés et associés aux transitions pour modéliser l'initialisation réelle ($init\_C2$ signifie que le chat est initialisé dans la pièce 2).



(a) Chat

(b) Modèle « Init » d'initialisation pour le chat et la souris

(c) Structure pour 3 modèles

FIGURE 28 – Modèles avec état initial global

Dans le but d'initialiser correctement les modèles et d'autoriser les évolutions uniquement lorsque tous les modèles ont été initialisés, un nouveau modèle est nécessaire. Ce modèle a une structure d'arbre, dont la taille dépend du nombre de modèles à initialiser. Chaque branche contient une combinaison d'événements qui initialisent tous les modèles. Enfin, le dernier état signifie que les modèles sont initialisés, par conséquent les différents événements y sont ajoutés (transitions rebouclantes) afin d'autoriser l'évolution des modèles.

Le modèle d'initialisation pour l'exemple du chat et de la souris est présenté dans la figure 28b. La structure du modèle d'initialisation pour un système comportant 3 modèles à initialiser est présentée dans la figure 28c. La structure pour n modèles peut être de grande dimension mais est facilement générable automatiquement.

Nous pouvons noter que le chat et la souris peuvent être initialisés dans la même pièce. Cela est normal car le modèle du système (G) doit représenter l'ensemble des états possibles du système même si certains sont dangereux. La suppression de ces états interdits, sera effectuée lors de la phase de synthèse du superviseur, à l'aide des modèles de spécifications.

Le superviseur résultant de ce modèle ($G = Chat||Souris||Init$) avec un état initial global est présenté dans la figure 29. Ce superviseur est bien évidemment de plus grande taille que le précédent (Fig. 27), mais il est valable pour tout état initial. De plus, les événements d'initialisation ($init\_Ri$ et $init\_Ci$) apparaissent dans le superviseur car ils sont définis comme contrôlables. Par conséquent, des conditions d'initialisation peuvent être extraites du superviseur pour des problèmes de reconfiguration (par exemple : il n'est pas autorisé

d'initialiser le chat dans la pièce 1 si la souris est déjà initialisé dans la pièce 3).

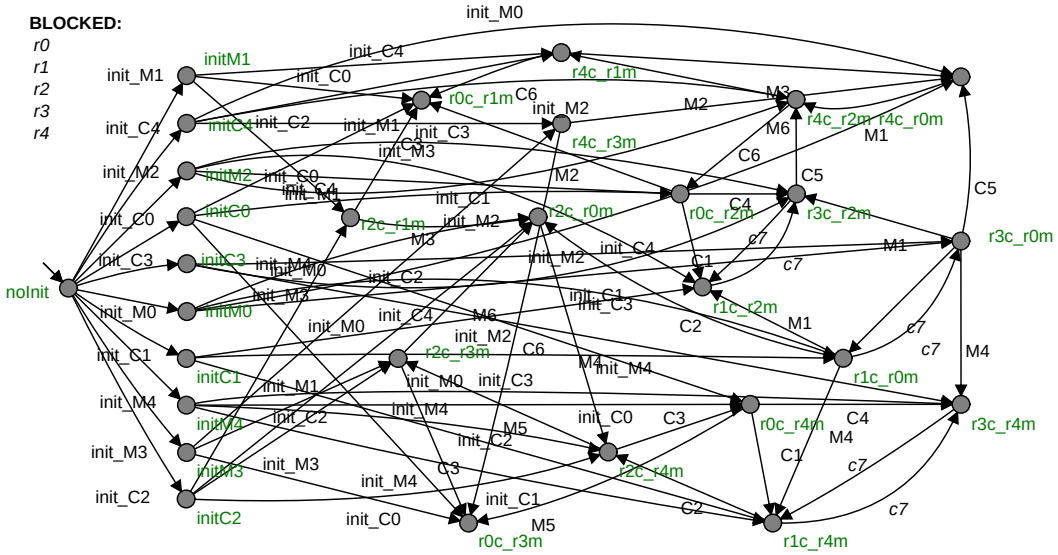


FIGURE 29 – Superviseur avec état initial global

II-3.3.2 Prise en compte de potentiels synchronismes d'événements

Comme discuté précédemment, l'hypothèse d'asynchronisme peut impliquer un deadlock du superviseur et donc du système contrôlé. De plus, si le superviseur est utilisé dans un automate programmable, les événements d'entrées et de sorties sont souvent synchronisés.

Dans l'approche SCT, deux événements synchronisés peuvent être modélisés par l'ajout d'un nouvel événement. Cet événement doit alors être ajouté de façon cohérente dans les modèles, afin de représenter le comportement synchrone. Dans cette partie, seule la synchronisation entre événements contrôlables est considérée, la synchronisation entre événements non-contrôlables est discutée plus tard dans la section III-5.3.

Pour l'exemple du chat et de la souris, les événements synchronisés sont créés et nommés Mi_Cj avec $(i, j) \in \{1, \ldots, 6\}^2$. Ces événements sont contrôlables car les deux événements Mi et Cj le sont. Puis ils sont ajoutés au modèle du chat et au modèle de la souris : sur chaque transition comportant Mi ou Cj est ajouté l'événement Mi_Cj. Le modèle de la souris résultant de l'ajout de ces nouveaux événements est présenté dans la figure 30.

Le superviseur calculé à partir de ces nouveaux modèles inclut le précédent (Fig. 29), mais comporte plus d'événements et de transitions (Tab. 11). Il est important de noter

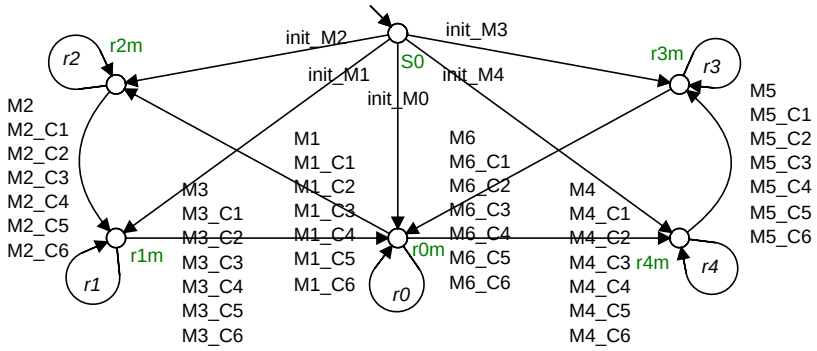


FIGURE 30 – Modèle de la souris avec événements synchrones et état initial global

que dans ce nouveau superviseur, il n'existe plus aucun état deadlock (par soucis de lisibilité ce superviseur n'est pas présenté).

II-3.3.3 Limitations de l'approche

L'ajout d'un état initial global permet de calculer un superviseur valable pour tous les états initiaux possibles. La modélisation des événements synchrones permet : la suppression d'états « deadlock » dans certains cas ; ainsi que la prise en compte de potentiels comportements synchrones du système, une fois celui-ci implémenté dans un automate programmable.

Malheureusement, les modèles sont de taille plus importante qu'avec l'approche de modélisation classique. Ceci implique une implémentation plus lourde et un risque d'explosion combinatoire (section I-3.3.2). Le tableau 11 illustre le fait que le nombre d'états et transitions augmente, lors de l'ajout de l'état initial global ou du mécanisme de synchronisation.

TABLE 11 – Comparaison quantitative des superviseurs

Superviseur	états	événements	transitions
1- init : $r4m.r2c$ et non synchro	7	18	11
2- init : $r4m.r2c$ et synchro	18	54	50
3- init : *globale* et non synchro	29	28	80
4- init : *globale* et synchro	29	64	96

Néanmoins, générer automatiquement l'état initial global pour chaque modèle, ainsi que le modèle d'initialisation présenté dans la section II-3.3.1 est possible sans difficulté. De même, il semble possible de générer automatiquement les événements synchronisés et de les ajouter de façon cohérente aux différentes transitions.

Cependant, même si la conception des modèles peut être aidée et guidée par des outils de génération automatique, les superviseurs résultant restent de grande taille et sont par conséquent compliqués à implémenter pour des systèmes réels complexes.

Une solution pour l'implémentation de ce superviseur dans un API, est d'utiliser l'approche de transformation du superviseur en contrôleur présentée précédemment (section II-2.5) (Cantarelli et Roussel, 2008). Néanmoins, cette approche (Cantarelli et Roussel, 2008) implique l'ajout de nouvelles étapes dans la conception du programme API (extraction du contrôleur, transformation en machine de Mealy, implémentation dans un langage API), ce qui complexifie un peu plus encore l'utilisation d'une telle approche.

II-4 Conclusion

Ce chapitre a présenté deux études permettant d'illustrer certaines des motivations présentées dans le chapitre I.

Dans un premier temps, la section II-2 a présentée une expérimentation menée auprès d'étudiants ingénieurs, de docteurs et de doctorants. Cette expérimentation a permis de mettre en évidence, un manque global de méthodologie et de formalisme, pour la conception d'un contrôleur.

Dans un second temps, la section II-3 a présenté comment la SCT peut être adaptée, pour compenser ses limitations lors de la conception d'un programme API (section I-3.3). Pour ce faire, une approche de modélisation a été proposée. Celle-ci permet la modélisation des événements synchrones ainsi que la prise en compte d'un état initial global. Néanmoins, cette approche implique un fort risque d'explosion combinatoire et ne permet pas de synthétiser directement un contrôleur, en effet il est tout d'abord nécessaire de synthétiser un superviseur.

Le chapitre suivant a pour objectif de proposer une approche formelle, utilisable dans l'industrie, pour la conception d'un programme API.

CHAPITRE III

Contribution à la commande par filtre logique pour les SED

III-1 Introduction

Les chapitres précédents ont montré que, pour la conception d'un programme API, le monde industriel n'utilise que très peu les méthodes formelles. Or, ces méthodes permettent de garantir la fiabilité d'un programme et de limiter le risque d'erreur lors de la conception. Il apparaît également que, les méthodes formelles « classiques » permettant la conception d'un contrôleur de systèmes à événements discrets, sont trop complexes à utiliser et mal adaptées aux APIs.

Ce chapitre III détaille l'approche par filtre logique (Fig. 31). Cette approche a pour but de proposer un compromis entre, le besoin d'approches formelles pour garantir la fiabilité des contrôleurs, et les contraintes liées au monde industriel telles que les habitudes de travail (cycle de développement) ou les obligations techniques (automate programmable industriel).

Tout d'abord, la section III-2 introduit le filtre logique et propose une méthodologie de conception. Ensuite, la section III-3 propose une formalisation des différents éléments du filtre logique. Puis, la section III-4 propose une approche de vérification formelle de la cohérence d'un ensemble de contraintes logiques. Enfin, la section III-5 permet de mettre en évidence des liens entre la SCT et l'approche par filtre logique.

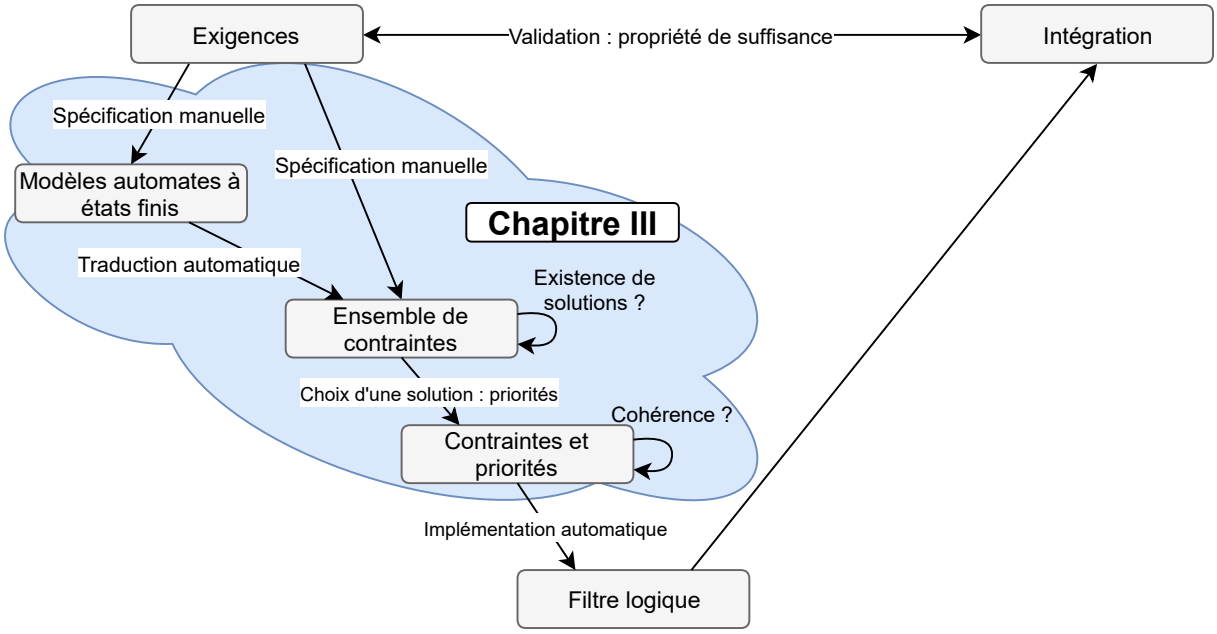


FIGURE 31 – Étapes du cycle de développement couvertes par le chapitre III

III-2 Méthodologie de conception du filtre logique

Cette section présente dans un premier temps le principe général d'un filtre logique et quelques définitions de vocabulaire (section III-2.1). Par la suite, la section III-2.2 propose une méthodologie globale permettant l'obtention d'un filtre logique. Enfin, la section III-2.4 présente les différentes utilisations possibles d'un filtre logique.

III-2.1 Principe et définitions

Les travaux précédents (Marangé, 2008; Riera *et al.*, 2015), présentés dans le chapitre I (section I-4.3), se sont focalisés sur la notion de **filtre de sécurité**, cette thèse propose de généraliser cette notion en parlant de **filtre logique**. Un **filtre logique** est un élément logiciel permettant de vérifier que des valeurs de variables respectent des exigences.

Ce travail se place dans un contexte de commande de système industriel commandé par des Automates Programmables Industriels (API). Dans un API, différents types de variables sont mis en jeu :

— les entrées (I) représentent en général les valeurs des capteurs du système contrôlé ;
— les sorties (O) sont les valeurs à envoyer aux actionneurs du système ;

— les variables internes permettant le stockage d'informations et les calculs.

Pour le filtre logique, deux ensembles de variables internes peuvent être identifiés : l'ensemble des variables à tester (G) et l'ensemble des observateurs logiques (E) (définis ci-après). Si le vecteur de variables G respecte toutes les exigences, alors le filtre logique « laisse passer » ces valeurs en les affectant aux actionneurs (vecteur O). Un **observateur logique** est une variable représentant une information non détectable directement par des capteurs. Par exemple, pour mémoriser la valeur d'une variable d'un cycle sur l'autre, ou bien encore pour savoir si une pièce est en cours de déplacement entre deux capteurs de position.

FIGURE 32 – Entrées/sorties du filtre logique

Dans les Systèmes à Événements Discrets (SED), la notion d'événement est prédominante dans les formalismes classiques (Ramadge et Wonham, 1989). Lorsque le but est le contrôle d'un SED, ces événements sont catégorisés en fonction d'un critère important : la contrôlabilité (Déf. 12). Dans la SCT, cette notion de contrôlabilité est à la base de l'approche de synthèse utilisée.

Définition 12. ***Contrôlabilité d'un événement*** *(Wonham, 2015)*
Un événement est contrôlable si et seulement si son occurrence peut être ***empêchée****.*

Cette notion de contrôlabilité permet de restreindre l'évolution du système, néanmoins elle ne permet pas d'imposer une évolution au système. En effet, empêcher un événement à apparaître n'implique pas que l'on soit capable de le forcer à apparaître. La notion de commandabilité permet de répondre à ce problème (Déf. 13). Les définitions sont proches mais néanmoins différentes, il semble donc important d'appuyer sur ce point essentiel, dès que l'on souhaite implémenter notre résultat dans un API.

Définition 13. ***Commandabilité d'un événement*** *(Brandin et Wonham, 1994)*
Un événement est commandable si et seulement si son occurrence peut être ***forcée****.*

Dans l'approche par filtre logique, la notion d'événement n'est pas au centre du formalisme. En effet dans un API, des variables sont utilisées et non des événements. Néanmoins, il existe une relation assez simple entre une variable logique et un événement : un événement représente un front montant ou descendant d'une variable. A chaque variable logique a, peuvent donc être associés deux événements distincts : le front montant $\uparrow a$ (passage de la valeur $faux$ à $vrai$) et le front descendant $\downarrow a$ (de $vrai$ à $faux$).

Dans cette thèse nous proposons de dériver les notions de contrôlabilité et de commandabilité d'un événement à une variable logique (Déf. 14 et 15).

Définition 14. ***Contrôlabilité d'une variable logique***
Une variable logique est contrôlable si et seulement si ses changements d'états (fronts montant et descendant) peuvent être ***empêchés****.*

Définition 15. ***Commandabilité d'une variable logique***
Une variable logique est commandable si et seulement si sa valeur logique peut être ***forcée****.*

Une variable commandable signifie donc que son front montant et son front descendant sont commandables, mais également qu'il est possible d'empêcher son changement d'état (contrôlabilité). Par conséquent, si une variable est commandable alors elle est contrôlable.

Dans la suite de ce manuscrit, les entrées du filtre (G, I, E) seront considérées comme non-commandables pour le filtre, les sorties (O) comme commandables (Fig. 32).

III-2.2 Obtention d'un filtre logique

Un filtre logique étant un élément logiciel ayant pour objectif d'être applicable dans un contexte industriel, nous proposons d'organiser les étapes de conception dans une approche

cyclique formelle (Fig. 33). La méthodologie proposée part des exigences, traditionnellement écrites en langage naturel, pour obtenir au final un programme implémentable facilement dans un Automate Programmable Industriel (API).

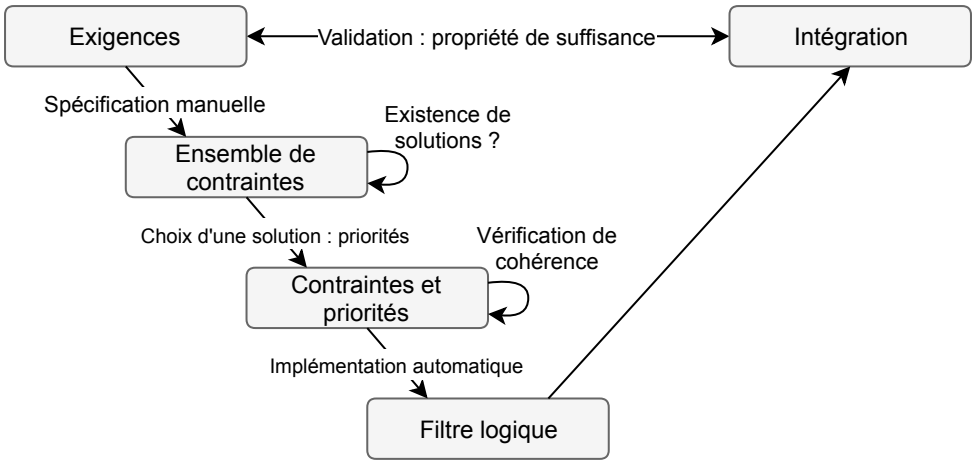


FIGURE 33 – Méthode cyclique formelle de conception d'un filtre logique

A partir des exigences décrites dans le cahier des charges du client, une étape de **formalisation** manuelle est nécessaire. Cette étape est très proche de celle utilisée dans la synthèse algébrique de Hietter (Hietter, 2009). Pour rappel, dans ces travaux sur la synthèse algébrique, il a été montré qu'il était possible de traduire des exigences écrites en langage naturel, vers un ensemble d'équations logiques appelées **contraintes** (Tab. 2, section I-4.2).

Dès lors qu'un ensemble de contraintes a été établi, des propriétés intrinsèques à cet ensemble peuvent être vérifiées formellement (existence, cohérence). La première propriété à vérifier est l'**existence de solutions** (vecteurs de sortie validant toutes les contraintes) quelle que soit la valeur des variables d'entrées du filtre.

Une fois cette première étape de vérification formelle effectuée, et dans le but d'obtenir une solution déterministe, le **choix d'une solution** particulière doit être effectué. La suite de ce chapitre (section III-3.3.2), montrera que ce choix revient à imposer des priorités entre les variables de sorties (actionneurs) de certaines contraintes logiques. Afin de vérifier formellement que les contraintes avec les priorités choisies sont cohérentes entre elles, nous proposons une approche de vérification formelle automatique nécessaire et suffisante.

Après ces deux étapes de conception, vérifiées formellement, l'implémentation du filtre logique est effectuée automatiquement (**génération automatique** du programme API).

L'implémentation du filtre logique se base sur un algorithme facilement intégrable dans un API en langage normalisé ST. Le chapitre IV présentera l'algorithme mis à jour lors de cette thèse, pour prendre en compte les évolutions de formalisations et de propriétés présentées dans ce chapitre III.

Certaines étapes de vérification, normalement présentes dans la partie montante du cycle en V, sont supprimées dans notre approche. Celles-ci sont intégrées formellement pendant la conception, afin de faciliter les tests exhaustifs. En effet, il est plus simple de tester des propriétés sur des modèles plutôt que sur un programme API.

La dernière étape est la **validation** du filtre logique. Contrairement à la vérification où le but est de tester des propriétés intrinsèques, la validation a pour but de tester des propriétés extrinsèques au filtre. Concrètement, la validation doit permettre de garantir que le filtre respecte les exigences du cahier des charges. Pour cela une propriété de suffisance a déjà été définie (Marangé, 2008).

III-2.3 Outil logiciel d'aide à la conception : SEDMA

Afin de faciliter l'utilisation du filtre logique et de la méthode de conception proposée précédemment, les différents traitements et algorithmes développés dans ce chapitre ont été implémentés dans l'outil logiciel SEDMA (Pichard *et al.*, 2017a).

SEDMA permet, à partir d'un fichier texte décrivant les contraintes logiques, d'appliquer les vérifications formelles proposées dans la suite de ce chapitre et de générer automatiquement le filtre logique dans un langage API (Structured Text).

Les détails du développement et de l'amélioration de ce logiciel ne sont pas traités dans cette thèse. Néanmoins, une présentation plus complète du logiciel et des fonctionnalités disponibles est disponible dans l'annexe B.

Une référence à SEDMA est effectuée dans la suite de ce manuscrit pour préciser quelles propositions ont été implémentées dans ce logiciel.

III-2.4 Utilisations du filtre logique

Le filtre logique étudié dans cette thèse peut être utilisé de différentes manières : le filtre de sécurité bloquant, le filtre de sécurité correcteur, le filtre fonctionnel et le filtre contrôleur.

Les filtres de sécurité (bloquant et correcteur) ont déjà été présentés dans le chapitre I (Fig. 17 et Fig. 20). Lors de la définition des contraintes (étape de formalisation), seules les exigences liées à la sécurité sont à traduire pour obtenir un filtre de sécurité. En effet, le but est uniquement de garantir que le contrôleur n'entrainera jamais le système dans un état ou un comportement interdit. Les équations logiques relatives à la partie sécurité du cahier des charges, c'est-à-dire ce que l'on ne doit pas pouvoir faire avec le système, sont appelées **contraintes de sécurité**.

Si l'on souhaite utiliser la partie fonctionnelle du cahier des charges, c'est-à-dire ce que l'on veut faire avec le système, les exigences relatives à cette partie sont alors traduites en équations logiques appelées **contraintes fonctionnelles**. Un **filtre fonctionnel** peut alors être synthétisé en suivant la méthode proposée précédemment (section III-2.2). Ce filtre permet de calculer un vecteur de sortie (valeurs actionneurs) respectant toutes les exigences fonctionnelles. Si le cahier des charges contient également des exigences de sécurité, un filtre de sécurité doit alors également être synthétisé.

D'un point de vue implémentation, le filtre de sécurité est placé après le filtre fonctionnel. Ce faisant, un vecteur de sortie respectant tout d'abord les exigences fonctionnelles est calculé, puis ce vecteur est testé et modifié si nécessaire par le filtre de sécurité (Fig. 34).

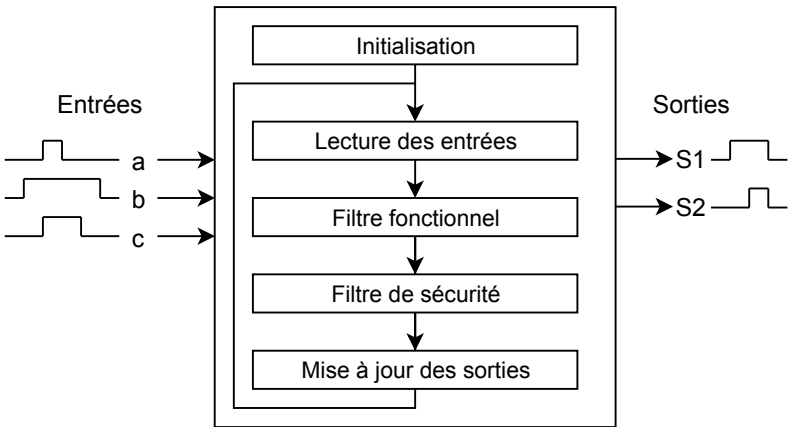


FIGURE 34 – Filtres en cascade

Cette approche de filtres en cascade permet une séparation claire des notions de sécurité

et de fonctionnel, ceci permet notamment une parallélisation des tâches de conception. Néanmoins, la séparation du fonctionnel et de la sécurité ne permet pas de garantir que toutes les exigences fonctionnelles seront toujours respectées. L'objectif de production ne peut donc pas être garanti.

Afin de palier à cet inconvénient, il est possible de mettre au même niveau les contraintes de sécurité et fonctionnelles. Un **filtre contrôleur** unique sera synthétisé à partir des exigences fonctionnelles et de sécurité. Dans ce cas, la propriété de cohérence des contraintes garantira qu'il existe toujours un vecteur de sortie respectant toutes les exigences.

Pour rappel, les travaux précédents ont introduit les contraintes de sécurité (Marangé, 2008). De plus, un premier travail présentant les contraintes fonctionnelles a également été proposé (Riera *et al.*, 2015). Bien que les noms de ces contraintes soient différents en fonction de leur utilisation, le formalisme associé est identique. Dans le cadre de ce chapitre III, nous proposons une formalisation plus générale des contraintes logiques. Le cadre formel et les outils associés proposés seront donc compatibles pour n'importe quel type de contraintes.

Dans la suite de ce chapitre, certaines étapes de la définition, de l'analyse et de l'utilisation du filtre logique sont illustrées à l'aide d'un exemple simple et complet.

III-2.5 Exemple du portail

Le système utilisé comme exemple est le même que celui utilisé dans le chapitre II. Pour rappel, le système présenté sur la figure 35 comporte 4 capteurs et 2 actionneurs (Tab. 12).

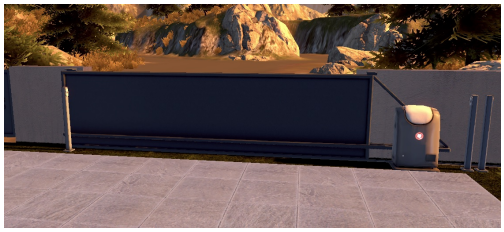


FIGURE 35 – Portail automatique

TABLE 12 – Capteurs et actionneurs

	type	commentaire
open	actionneur	ordre d'ouverture
close	actionneur	ordre de fermeture
go	capteur	porte ouverte
gc	capteur	porte fermée
remote	capteur	bouton télécommande
car	capteur	détection d'une voiture

Le cahier des charges est légèrement différent de celui utilisé précédemment dans le chapitre II :

— La porte doit s'ouvrir lorsque le bouton de la télécommande est activée.
— La porte doit se fermer uniquement s'il n'y a pas de voiture sur la trajectoire (capteur *car*) après une temporisation.
— La sécurité des équipements et des usagers doit toujours être garantie.

Pour cet exemple, seule la partie sécurité du cahier des charges sera considérée. Par conséquent, le résultat sera un filtre de sécurité. Néanmoins, le formalisme, les propriétés et les outils présentés dans la suite de ce chapitre seront valables pour n'importe quel type de filtre logique.

III-3 Formalisation du filtre logique

Dans cette partie, le formalisme lié au filtre logique est détaillé. Dans un premier temps, une définition du filtre et des notations associées sont proposées. Puis, les contraintes logiques et les fonctions associées sont définies. Ensuite, la signification et l'utilisation d'une contrainte sont présentées. Enfin, des priorités entre les variables commandables d'une contrainte sont définies pour rendre déterministe la résolution.

Un filtre logique est composé de littéraux, de contraintes logiques et de priorités fonctionnelles (Déf. 16).

Définition 16. *Un **filtre logique** est un triplet* $\langle \overset{*}{L}, C, P \rangle$ *tel que :*
— $\overset{*}{L} = \overset{*}{Y} \cup \overset{*}{O}$ *est un ensemble de Littéraux.*
— C *est un ensemble de Contraintes logiques.*
— P *est un ensemble de Priorités fonctionnelles.*
Avec, $\overset{*}{Y}$ *l'ensemble des littéraux non-commandables et* $\overset{*}{O}$ *l'ensemble des littéraux commandables pour le filtre.*

Un littéral $\overset{*}{l} \in \overset{*}{L}$ est une variable logique l ou son complément $\bar{l}$. L'ensemble des littéraux utilisés par un filtre logique peut être séparé en deux catégories. L'ensemble des littéraux non-commandables est noté $\overset{*}{Y}$ et l'ensemble des littéraux commandables est noté $\overset{*}{O}$. Le

nombre de littéraux commandables est noté $N_o = dim(\overset{*}{O})$. Le nombre de contraintes est noté $N_c = dim(C)$.

Dans cette thèse, les capteurs du système (entrées du contrôleur) seront considérés comme non-commandables. Les actionneurs du système (sorties du contrôleur) seront par contre considérés comme commandables.
Enfin, vis à vis du filtre logique, les variables internes du contrôleur, calculées en amont du filtre, s'ajoutent à l'ensemble des capteurs pour former l'ensemble $\overset{*}{Y}$. L'ensemble des littéraux commandables du filtre $\overset{*}{O}$ est constitué uniquement des actionneurs.

III-3.1 Définition des contraintes logiques

Dans l'approche proposée, les exigences sont traduites sous forme d'équations booléennes appelées contraintes logiques. Une contrainte logique est définie comme un monôme logique, c'est-à-dire un produit de littéraux.

Il existe deux types de littéraux (commandables et non-commandables), une contrainte logique est donc composée d'une partie non-commandable et d'une partie commandable. Une **contrainte logique** $C_n \in C$ est une fonction qui, à un sous-ensemble de littéraux non-commandables et un sous-ensemble de littéraux commandables associe une fonction booléenne (Déf. 17).

Définition 17. *Soient $\overset{*}{Y}^{C_n} \in 2^{\overset{*}{Y}}$ l'ensemble des littéraux non-commandables de C_n, $\overset{*}{O}^{C_n} \in 2^{\overset{*}{O}}$ l'ensemble des littéraux commandables de C_n et Γ l'ensemble des fonctions booléennes définies dans Hietter (2009), alors la contrainte logique C_n est définie de la façon suivante :*

$$C_n : \begin{array}{ccc} 2^{\overset{*}{Y}} \times 2^{\overset{*}{O}} & \longrightarrow & \Gamma \\ (\overset{*}{Y}^{C_n}, \overset{*}{O}^{C_n}) & \longmapsto & \prod(\overset{*}{Y}^{C_n}) \cdot \prod(\overset{*}{O}^{C_n}) = Y^{C_n} \cdot O^{C_n} \end{array} , \forall n \in \{1, \ldots, N_c\} \tag{III.1}$$

Avec Y^{C_n} le monôme logique non-commandable de C_n et O^{C_n} le monôme logique commandable de C_n.

Dans les travaux précédents (Marangé, 2008; Riera *et al.*, 2015), deux types de contraintes

logiques étaient proposés : les contraintes simples et les contraintes combinées. Néanmoins, la définition de ces contraintes rendait difficile leur généralisation. Dans cette thèse, une redéfinition des types de contraintes est proposée (Déf. 18). De plus nous définissons un troisième type : les contraintes structurelles. Ces dernières permettront de modéliser des informations structurelles du système, utilisées lors des étapes de vérification formelle du filtre logique.

Définition 18. *Le **type** d'une contrainte logique dépend du nombre de littéraux commandables que contient cette contrainte.*

- *Une contrainte est dite **structurelle** si elle ne contient aucun littéral commandable. L'ensemble des contraintes structurelles est noté $Cstruct$. Le nombre de contraintes structurelles est noté $N_{Cstruct}$.*
- *Une contrainte est dite **simple** si elle contient un seul littéral commandable. L'ensemble des contraintes simples est noté Cs. Le nombre de contraintes simples est noté N_{Cs}.*
- *Une contrainte est dite **combinée** si elle contient plusieurs littéraux commandables. L'ensemble des contraintes combinées est noté Cc. Le nombre de contraintes combinées est noté N_{Cc}.*

Par conséquent, le nombre de contraintes est $N_c = N_{Cstruct} + N_{Cs} + N_{Cc}$. La suite de cette partie détaille la définition d'une contrainte (Déf. 17) en fonction de son type.

Contraintes structurelles Une contrainte structurelle, notée par la suite $Cstruct_{n_1}$, est une contrainte portant uniquement sur des littéraux non-commandables (Déf. 19).

Définition 19. *Soit $Y^{\overset{*}{Cstruct_{n_1}}} \in 2^{\overset{*}{Y}}$ l'ensemble des littéraux non-commandables de $Cstruct_{n_1}$, alors :*

$$Cstruct_{n_1} : \begin{array}{ccc} 2^{\overset{*}{Y}} & \longrightarrow & \Gamma \\ Y^{\overset{*}{Cstruct_{n_1}}} & \longmapsto & \prod(Y^{\overset{*}{Cstruct_{n_1}}}) \end{array}, \forall n_1 \in \{1, \ldots, N_{Cstruct}\} \quad \text{(III.2)}$$

Contraintes simples Une contrainte simple est une contrainte contenant un seul littéral commandable (Déf. 20).

Définition 20. *Soient* $\overset{*}{Y}^{Cs_{n_2}} \in 2^{\overset{*}{Y}}$ *l'ensemble des littéraux non-commandables de* Cs_{n_2} *et* $\overset{*}{o}_k \in \overset{*}{O}$ *le k-ième littéral commandable. Alors, une contrainte simple sur* $\overset{*}{o}_k$ *est :*

$$Cs^k_{n_2} : \begin{array}{ccc} 2^{\overset{*}{Y}} \times \overset{*}{O} & \longrightarrow & \Gamma \\ (\overset{*}{Y}^{Cs_{n_2}}, \overset{*}{o}_k) & \longmapsto & \prod(\overset{*}{Y}^{Cs_{n_2}}) \cdot \overset{*}{o}_k \end{array}, \forall n_2 \in \{1, \ldots, N_{Cs}\}, \forall k \in \{1, \ldots, N_o\} \tag{III.3}$$

Contraintes combinées Une contrainte combinée, notée par la suite $Cc_{n_3} \in Cc$, est une contrainte contenant plusieurs littéraux commandables.

Définition 21. *Soient* $\overset{*}{Y}^{Cc_{n_3}} \in 2^{\overset{*}{Y}}$ *l'ensemble des littéraux non-commandables de* Cs_{n_3} *et* $\overset{*}{O}^{Cc_{n_3}} \in 2^{\overset{*}{O}}$ *l'ensemble des littéraux commandables de* Cs_{n_3} *avec* $dim(\overset{*}{O}^{Cc_{n_3}}) > 1$*, alors :*

$$Cc_{n_3} : \begin{array}{ccc} 2^{\overset{*}{Y}} \times 2^{\overset{*}{O}} & \longrightarrow & \Gamma \\ (\overset{*}{Y}^{Cc_{n_3}}, \overset{*}{O}^{Cc_{n_3}}) & \longmapsto & \prod(\overset{*}{Y}^{Cc_{n_3}}) \cdot \prod(\overset{*}{O}^{Cc_{n_3}}) \end{array}, \forall n_3 \in \{1, \ldots, N_{Cc}\} \tag{III.4}$$

III-3.2 Utilisation des contraintes

Dans l'approche par filtre logique, une contrainte est donc un produit logique de littéraux. Il est possible de définir une contrainte satisfaite (Déf. 22) et une contrainte violée (Déf. 23).

Définition 22. *Une contrainte logique est dite* ***satisfaite*** *lorsque celle-ci est évaluée à* $faux$ *(Éq. III.5).*

$$C_n = \prod(\overset{*}{Y}^{C_n}) \cdot \prod(\overset{*}{O}^{C_n}) = 0 \tag{III.5}$$

Définition 23. *Une contrainte logique est dite* ***violée*** *lorsque celle-ci est évaluée à vraie (Éq. III.6).*

$$C_n = \prod (\overset{*}{Y}^{C_n}) \cdot \prod (\overset{*}{O}^{C_n}) = 1 \tag{III.6}$$

Par conséquent, dans le but d'être satisfaite, une contrainte peut être lue de la manière suivante : « Si la partie non-commandable d'une contrainte est évaluée à $vrai$, alors sa partie commandable doit être évaluée à $faux$ ». Durant l'étape de spécification, permettant de traduire les exigences informelles en contraintes logiques, chaque type de contrainte permet de traduire un type d'exigence :

— les contraintes simples permettent d'exprimer des exigences de type « si ... alors ... » ;
— les contraintes combinées permettent d'exprimer des exigences d'exclusivité entre variables commandables ;
— les contraintes structurelles permettent d'exprimer une exclusivité entre variables non-commandables. Celles-ci ne sont pas utilisées par le filtre logique mais permettent de réduire l'espace d'état à explorer pendant la vérification formelle des contraintes.

Le but d'un filtre logique est de garantir que les valeurs des variables commandables permettent de satisfaire l'ensemble des exigences, et donc, l'ensemble des contraintes logiques. Par conséquent, une condition de satisfiabilité d'un ensemble de contraintes logiques est définie (Déf. 24), cette condition permet de tester rapidement la validité de l'ensemble des contraintes.

Définition 24. *Un ensemble de contraintes est* ***satisfait*** *si toutes les contraintes sont satisfaites. Ceci revient à vérifier l'équation suivante :*

$$\sum_{k=1}^{N_o} \sum_{i=1}^{Ncs^k} Cs_i^k + \sum_{j=1}^{Ncc} Cc_j = 0 \tag{III.7}$$

Comme présenté précédemment, les contraintes logiques sont utilisées pour vérifier que les valeurs des variables logiques satisfont les exigences. La suite de ce chapitre va présenter

comment il est possible de résoudre les contraintes logiques violées.

Portail 1. ***Définition des contraintes***

Pour l'exemple du portail, un littéral est associé à chaque information logique (capteurs, actionneurs...). De plus, l'observateur logique $\overset{*}{open}^{-1}$ *est créé (variable interne).* $\overset{*}{open}^{-1}$ *est égal à* $\overset{*}{open}$ *au cycle précédent (mémorisation).*

- $\overset{*}{I} = \{\overset{*}{go}, \overset{*}{gc}, \overset{*}{remote}, \overset{*}{car}\}$
- $\overset{*}{E} = \{\overset{*}{open}^{-1}\}$
- $\overset{*}{Y} = \overset{*}{I} \cup \overset{*}{E}$
- $\overset{*}{O} = \{\overset{*}{open}, \overset{*}{close}\} = \{o_1, o_2\}$

Pour cet exemple du portail, l'objectif est de mettre en œuvre un filtre de sécurité. Par conséquent, les parties fonctionnelles du cahier des charges n'ont pas besoin d'être considérées pour la définition des contraintes. Une ré-écriture du cahier des charges, décrit dans la section III-2.5, est proposée ci-dessous.

- *Partie fonctionnelle :*
  - *Lors de l'appui sur le bouton de la télécommande, le portail doit s'ouvrir.*
  - *Le portail doit se refermer après une temporisation.*
- *Partie sécurité :*
  - *1- Les efforts inutiles sur le moteur doivent être empêchés.*
  - *2- Les collisions doivent être évitées.*

A partir de la partie sécurité du cahier des charges, décrite ci-dessus, les contraintes peuvent être définies :

Contraintes simples :

- $Cs_1^1 = go \cdot \overline{gc} \cdot open \equiv$ *si le portail est totalement ouvert, alors l'ordre d'ouverture ne doit pas être envoyé.*
- $Cs_2^1 = \overline{go} \cdot \downarrow open = \overline{go} \cdot open^{-1} \cdot \overline{open} \equiv$ *Tant que le portail n'est pas complètement ouvert, l'ouverture ne doit pas être interrompue.*
- $Cs_3^1 = car \cdot \overline{gc} \cdot \overline{go} \cdot \overline{open} \equiv$ *si une voiture est détectée (car = 1) et si le portail n'est ni fermé (gc = 0) ni ouvert (go = 0) entièrement, alors le portail doit s'ouvrir (open = 1).*
- $Cs_1^2 = car \cdot close \equiv$ *si une voiture est détectée, alors le portail ne*

doit pas se fermer.

— $Cs_2^2 = gc \cdot \overline{go} \cdot close \equiv$ *si le portail est totalement fermé, l'ordre de fermeture ne doit pas être envoyé.*

Contrainte combinée :

— Cc_1 *:* $open \cdot close \equiv$ *les ordres d'ouverture et de fermeture ne doivent pas être actifs simultanément.*

Contrainte structurelle :

— C_{struct_1} *:* $go \cdot gc \equiv$ *les capteurs de fin de course du portail ne peuvent pas être actifs en même temps.*

Les contraintes $\{Cs_1^1, Cs_2^2, Cc_1\}$ *sont relatives au point 1 de la partie sécurité du cahier des charges. Les contraintes* $\{Cs_2^1, Cs_3^1, Cs_1^2\}$ *sont relatives au point 2. La contrainte structurelle* C_{struct_1} *ne sera utilisée que dans l'étape d'analyse du filtre (Section III-4).*

III-3.3 Résolution d'une contrainte

Une contrainte violée doit être résolue afin de garantir que l'exigence associée soit respectée à la fin du cycle automate. Pour résoudre une contrainte violée, les valeurs de ses variables commandables doivent être changées. Il existe deux types de contrainte faisant apparaître des variables commandables (Cs et Cc), la manière de les résoudre n'est donc pas la même.

III-3.3.1 Résolution d'une contrainte simple

La seule façon de résoudre une contrainte simple est de complémenter la valeur de sa variable commandable. La résolution d'une Cs violée est donc déterministe structurellement.

Plusieurs contraintes peuvent exister pour un même littéral commandable $\overset{*}{o}_k \in \overset{*}{O}$ ($k^{\text{ième}}$ littéral de $\overset{*}{O}$). Le nombre de contraintes simples sur $\overset{*}{o}_k$ est noté Ncs^k. Étant donné qu'un littéral $\overset{*}{v}$ est une variable logique v ou sa négation $\overline{v}$, il existe deux formes distinctes de contraintes simples.

Considérant $i \in [1, Ncs^k]$, $k \in [1, N_o]$, $Y0_i^k \in 2^Y$ et $Y1_i^k \in 2^Y$, alors la $i^{\text{ième}}$ contrainte simple de $\overset{*}{o_k}$ (Cs_i^k) est définie Éq. (III.8).

$$Cs_i^k = \begin{cases} \prod(Y0_i^k) \cdot o_k \\ OU \\ \prod(Y1_i^k) \cdot \overline{o_k} \end{cases} \tag{III.8}$$

Ncs_1^k est le nombre de contraintes simples contenant o_k et Ncs_2^k est le nombre de contrainte simple contenant $\overline{o_k}$. Donc, il existe $Ncs^k = Ncs_1^k + Ncs_2^k$ contraintes simples sur la variable commandable o_k.

En combinant chaque contrainte simple de o_k, il est possible de définir deux fonctions polynomiales logiques $F0s^k$ et $F1s^k$ (Éq. (III.9)). Ces fonctions sont composées uniquement des variables non-commandables présentes dans les contraintes simples associées à $\overset{*}{o_k}$ (o_k ou $\overline{o_k}$).

$$\begin{aligned} \sum_{i=1}^{Ncs^k} Cs_i^k &= \sum_{i=1}^{Ncs^k} (\prod(Y0_i^k) \cdot o_k + \prod(Y1_i^k) \cdot \overline{o_k}) \\ &= \sum_{i=1}^{Ncs_1^k} (\prod(Y0_i^k)) \cdot o_k + \sum_{i=1}^{Ncs_2^k} (\prod(Y1_i^k)) \cdot \overline{o_k} \\ \sum_{i=1}^{Ncs^k} Cs_i^k &= F0s^k \cdot o_k + F1s^k \cdot \overline{o_k} \end{aligned} \tag{III.9}$$

En considérant les Éq. (III.9) et (III.7), l'équation (III.10) doit être vérifiée pour garantir que toutes les contraintes simples sont résolues.

$$F0s^k \cdot o_k + F1s^k \cdot \overline{o_k} = 0 \tag{III.10}$$

Par conséquent, les fonctions $F0s^k$ et $F1s^k$ indiquent si la variable commandable o_k doit être forcée à 0 ou 1. En effet, si $F0s^k = 1$, alors o_k doit être forcée à 0 afin de vérifier l'équation (III.10) (réciproquement à 1 pour $F1s^k$).

Définition 25. ***Fonctions de forçage simples***

$F0s^k$ est la fonction de forçage à 0 simple de o_k et $F1s^k$ est la fonction de forçage à 1 simple de o_k.

L'extraction automatique des fonctions de forçage simples, à partir des contraintes simples, a été implémentée dans SEDMA (Annexe II-3.3).

Portail 2. ***Extraction des fonctions*** $F0s^k$ ***et*** $F1s^k$

$$\textit{open (i.e. } o_1\textit{)}: \begin{cases} F0s^1 = go \cdot \overline{gc} & (\textit{extrait de } Cs_1^1) \\ F1s^1 = \overline{go} \cdot open^{-1} + car \cdot \overline{gc} \cdot \overline{go} & (Cs_2^1 \textit{ et } Cs_3^1) \end{cases}$$

$$\textit{close (i.e. } o_2\textit{)}: \begin{cases} F0s^2 = car + gc \cdot \overline{go} & (Cs_1^2 \textit{ et } Cs_2^2) \\ F1s^2 = 0 \end{cases}$$

III-3.3.2 Résolution d'une contrainte combinée

Si une contrainte combinée (Cc) est violée, un choix doit être fait pour la résoudre. Ce choix dépend du nombre de variables commandables contenues dans cette contrainte. En effet, pour une Cc avec 2 variables commandables, 3 possibilités existent pour la résoudre : complémenter la valeur de 1 variable uniquement ou bien complémenter les deux en même temps. Par conséquent, pour une Cc contenant n variables commandables, une solution doit être choisie parmi les $2^n - 1$ possibilités.

La résolution d'une Cc n'est donc pas déterministe. Dans le but de pouvoir maîtriser le comportement du système, la résolution doit être déterministe. cette section introduit une notion de priorité entre les variables commandables d'une contrainte combinée.

En-ligne, étant donnée une affectation des variables non-commandables qui viole au moins une contrainte combinée (vecteur G dans la figure 32), la solution finale calculée par le filtre logique doit être toujours la même. Si cela est le cas, la résolution est déterministe. Pour cela, hors-ligne, chaque contrainte combinée doit être associée à une priorité. Cette priorité permet d'indiquer quelles variables commandables sont prioritaires par rapport aux autres dans cette contrainte combinée.

Dans les travaux précédents sur le filtre logique (Coupat, 2014; Riera *et al.*, 2015), cette

notion de priorité était traduite à l'aide de différentes matrices. Cette modélisation n'était valable que dans le cas d'une contrainte combinée avec 2 variables commandables.

Par conséquent, dans cette thèse nous définissons deux niveaux de priorités. La première dépend uniquement de la structure de la contrainte combinée, c'est-à-dire le nombre de variables commandables. La seconde est relative à un choix à faire durant la phase de définition des contraintes. Avec cette formalisation des priorités, l'utilisation de contraintes combinées de taille quelconque est rendue possible.

Priorité structurelle Pour une Cc avec deux variables commandables, si l'une d'elles est forcée par la résolution d'une Cs, alors la variable commune est temporairement non-commandable pour cette Cc. Par conséquent, il ne reste qu'une seule variable commandable, cette Cc est donc temporairement une Cs et le choix pour la résoudre est déterministe : complémenter la valeur de la variable commandable.

Étant donné que cette indication de forçage par une Cs est traduite à l'aide des fonctions $F0s$ et $F1s$, il est possible de traduire cette priorité structurelle à l'aide d'une équation logique.

Portail 3. ***Priorité structurelle***

Sur le système du portail seule la contrainte combinée $Cc_1 = o_1 \cdot o_2 = open \cdot close$ *existe. Celle-ci implique que, si open est forcé à* 1 *par une contrainte simple* ($F1s^1 = 1$)*, alors la seule façon de résoudre la combinée* Cc_1 *est de forcer close à* 0*. Ceci se traduit par l'équation suivante :*

$$open \cdot F1s^2 + F1s^1 \cdot close = 0$$

Cette transformation en équation logique peut être étendue à une contrainte combinée avec plus de deux variables commandables.

Exemple 1. *Soit a une variable non-commandable,* $\{O_1, O_2, O_3\}$ *trois variables commandables et* $Cc_2 = a \cdot O_1 \cdot O_2 \cdot \overline{O_3}$ *une contrainte combinée. Alors l'équation*

à vérifier pour Cc_2 est :

$$a \cdot (O_1 \cdot F1s^2 \cdot F0s^3 + F1s^1 \cdot O_2 \cdot F0s^3 + F1s^1 \cdot F1s^2 \cdot \overline{O_3}) = 0$$

Ce mécanisme est formalisé en définissant la notion de priorité structurelle (Déf. 26).

Définition 26. ***Priorité structurelle***

Considérons un sous-ensemble de littéraux non-commandables $\overset{*}{Y}^{Cc} \subseteq \overset{*}{Y}$, un sous-ensemble de littéraux commandables $\overset{*}{O}^{Cc} \subseteq \overset{*}{O}$ et la contrainte combinée sur ces littéraux $Cc = \prod(\overset{*}{Y}^{Cc}) \cdot \prod(\overset{*}{O}^{Cc})$. Alors la forme générale de l'équation traduisant la priorité structurelle associée à Cc est définie ci-dessous avec :

$$Fs^j = \begin{cases} F0s^j \ si\ \overline{o_j} \in Cc \\ F1s^j \ si\ o_j \in Cc \\ 0 \ sinon \end{cases} \quad et\ \overset{*}{o}_k = \begin{cases} o_k \ si\ o_k \in Cc \\ \overline{o_k} \ si\ \overline{o_k} \in Cc \\ 0 \ sinon \end{cases}$$

$$prioS_{Cc} : \prod(\overset{*}{Y}^{Cc}) \cdot \sum_{k=1}^{dim(\overset{*}{O})} (\overset{*}{o}_k \cdot \prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (Fs^j)) = 0 \qquad \text{(III.11)}$$

Cette équation peut être développée :

$$prioS_{Cc} : \sum_{k=1}^{dim(\overset{*}{O})} (\prod(\overset{*}{Y}^{Cc}) \cdot \prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (Fs^j) \cdot \overset{*}{o}_k) = 0 \qquad \text{(III.12)}$$

Enfin, des monômes peuvent être définis :

$$\forall k \mid \overset{*}{o}_k \in Cc, prioS^k_{Cc} = \prod(\overset{*}{Y}^{Cc}) \cdot \prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (Fs^j) \cdot \overset{*}{o}_k \qquad \text{(III.13)}$$

Chaque monôme $prioS^k_{Cc}$ est composé d'une partie non-commandable ($\prod(\overset{*}{Y}^{Cc}) \cdot \prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (Fs^j)$) noté $\overset{*}{Y}^{prioS^k_{Cc}}$ et d'une variable commandable (o_k ou $\overline{o_k}$). Un monôme traduit donc une condition suffisante au forçage de $\overset{*}{o}_k$, dans le cas où toutes les autres variables commandables d'une contrainte combinée sont forcées par des contraintes simples.

Priorité fonctionnelle Quand au moins deux variables commandables restent libres (non forcées) dans une contrainte combinée, il existe plusieurs solutions pour résoudre

cette contrainte. L'exemple 4 illustre les différentes possibilités dans le cas d'une contrainte combinée à 2 variables commandables.

Portail 4. ***Priorité possible pour le portail***

L'exemple du portail ne comporte qu'une seule contrainte combinée ($Cc_1 = open \cdot close$), par conséquent un seul choix est à faire parmi les suivants :

1- $Cc_1 = 1 \implies close := 0$ *(priorité à open)*

2- $Cc_1 = 1 \implies open := 0$ *(priorité à close)*

3- $Cc_1 = 1 \implies open := 0$ *ET* $close := 0$ *(pas de priorité)*

Ces trois possibilités permettent de résoudre la Cc_1, néanmoins la première semble « meilleure » par rapport à l'utilisation normale d'un système de portail. En effet, il peut être supposé que si les deux ordres sont envoyés en même temps (portail en cours de fermeture et appui sur bouton de la télécommande), alors l'utilisateur préfère ouvrir le portail.

Dans ces conditions, un choix doit avoir été fait lors de la phase de définition des contraintes. Ce choix, appelé priorité fonctionnelle, va permettre de résoudre de manière déterministe la contrainte combinée en forçant une ou plusieurs variables commandables. Comme illustré dans l'exemple 4, ce choix est en général lié à l'utilisation finale du système (partie fonctionnelle du cahier des charges).

De la même façon que pour les priorités structurelles, il est possible de définir une équation générale pour la priorité fonctionnelle associée à une contrainte combinée (Déf. 27). Cette équation booléenne doit permettre d'indiquer quelle variable doit être forcée dans la combinée lorsque celle-ci est violée et qu'au moins deux variables restent libres.

Définition 27. ***Priorité fonctionnelle***

Considérons un sous-ensemble de littéraux non-commandables $\overset{*}{Y}^{Cc} \subseteq \overset{*}{Y}$*, un sous-ensemble de littéraux commandables* $\overset{*}{O}^{Cc} \subseteq \overset{*}{O}$ *et la contrainte combinée* $Cc = \prod(\overset{*}{Y}^{Cc}) \cdot \prod(\overset{*}{O}^{Cc})$*. Enfin,* $\overset{*}{G}^{Cc}$ *est l'ensemble des littéraux commandables à tester.*

Pour toute priorité fonctionnelle telle que « $Cc = 1 \implies$ *forcer* $\overset{*}{o}_k$ *», un monôme logique est défini :*

$$prioF^k_{Cc} = \prod(\overset{*}{Y}^{Cc}) \cdot \prod(\overset{*}{G}^{Cc}) \cdot \prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (\overline{Fs^j}) \cdot \overset{*}{o}_k \tag{III.14}$$

$$\text{avec } Fs^j = \begin{cases} F0s^j \ si\ \overline{o_j} \in Cc \\ F1s^j \ si\ o_j \in Cc \\ 0 \ sinon \end{cases} \quad \text{et } \overset{*}{o}_k = \begin{cases} o_k \ si\ o_k \in Cc \\ \overline{o_k} \ si\ \overline{o_k} \in Cc \\ 0 \ sinon \end{cases}$$

Si aucune priorité fonctionnelle ne concerne $\overset{*}{o}_k$ *pour la contrainte* Cc*, alors* $prioF^k_{Cc} = 0$*.*

Rappel : les littéraux non-commandables $\overset{*}{G}$ *correspondent aux variables commandables à tester (Fig. 32).* $\overset{*}{G}^{Cc}$ *correspond donc à la restriction de* $\overset{*}{G}$ *aux littéraux commandables de* Cc *(*$\overset{*}{O}^{Cc}$*).*

Tout comme pour les priorités structurelles, une priorité fonctionnelle (Éq. III.14) est traduite à l'aide d'un monôme logique. De plus, chaque monôme est composé d'une partie non-commandable notée $\overset{*}{Y}^{prioF^k_{Cc}}$ et d'une variable commandable. La partie non-commandable est détaillée ci-dessous :

- $\prod(\overset{*}{Y}^{Cc}) \cdot \prod(\overset{*}{G}^{Cc})$: permet de savoir si la contrainte Cc est violée par le vecteur à tester (G).
- $\prod_{j=1}^{dim(\overset{*}{O}),j\neq k} (\overline{Fs^j})$: permet de vérifier qu'aucune contrainte simple ne force les variables commandables de Cc.

Portail 5. ***Priorité fonctionnelle choisie et traduction en équation***

Considérons la contrainte Cc_1 : $open \cdot close = 0$*. Pour rappel,* $open = o_1$ *et* $close = o_2$*. Si la priorité fonctionnelle choisie est la suivante :* «$Cc_1 = 1 \implies$

close $= 0$ » *(priorité à open), alors le monôme traduisant cette priorité est le suivant :*

$$prioF^1_{Cc_1} = 0$$

$$prioF^2_{Cc_1} = (g_1 \cdot g_2) \cdot (\overline{F1s^1} \cdot \overline{F1s^2}) \cdot close$$

Si la priorité fonctionnelle choisie est la suivante : « $Cc_1 = 1 \implies open := 0$ *ET* $close := 0$ » *(pas de priorité), alors les monômes sont :*

$$prioF^1_{Cc_1} = (g_1 \cdot g_2) \cdot (\overline{F1s^1} \cdot \overline{F1s^2}) \cdot open$$

$$prioF^2_{Cc_1} = (g_1 \cdot g_2) \cdot (\overline{F1s^1} \cdot \overline{F1s^2}) \cdot close$$

La résolution d'une contrainte combinée est plus difficile que la résolution d'une contrainte simple. En effet, pour une contrainte simple, seule la structure de cette contrainte est nécessaire pour savoir quelle variable doit être forcée pour la résolution. Pour une contrainte combinée, il est nécessaire d'effectuer un choix de résolution arbitraire (priorité fonctionnelle), mais également de prendre en compte les informations liées à la résolution des contraintes simples ($F0s$ et $F1s$) via les priorités structurelles.

L'extraction automatique des priorités structurelles, à partir des contraintes combinées, a été implémentée dans SEDMA (Annexe II-3.3). De même, les priorités fonctionnelles sont traduites automatiquement à partir du fichier d'entrée fourni à SEDMA (description des contraintes et des priorités fonctionnelles).

Tout comme pour les contraintes simples, nous proposons de définir des fonctions de forçage combinées.

Fonctions de forçage combinées Il est possible de définir des fonctions de forçage, permettant de savoir si une variable commandable doit être forcée à 0 ou à 1, pour résoudre une contrainte combinée en respectant les priorités structurelles et fonctionnelles.

La construction des fonctions de forçage combinées ($F0c$ et $F1c$) est analogue à celle des fonctions simples. Pour rappel, ces dernières sont construites par extraction des parties non-commandables des contraintes simples de o_k (pour $F0s^k$) et de $\overline{o_k}$ (pour $F1s^k$). Néanmoins, dans le cas des fonctions combinées, ce sont les monômes traduisant les priorités (structurelles et fonctionnelles) qui sont utilisés et non pas les contraintes directement. Une définition formelle de ces fonctions est proposée ci-dessous (Déf. 28).

Définition 28. ***Fonctions de forçage combinées***

La fonction de forçage à 0 combinée de $\overset{*}{o}_k$ *est définie comme : la somme logique des parties non-commandables des monômes traduisant les priorités structurelles et fonctionnelles concernant* o_k*.*

$$\forall Cc \mid o_k \in Cc, F0c^k = \sum_{Cc} (Y^{prioS^k_{Cc}} + Y^{prioF^k_{Cc}}) \qquad \text{(III.15)}$$

La fonction de forçage à 1 combinée de $\overset{*}{o}_k$ *est définie comme : la somme logique des parties non-commandables des monômes traduisant les priorités structurelles et fonctionnelles concernant* $\overline{o_k}$*.*

$$\forall Cc \mid \overline{o_k} \in Cc, F1c^k = \sum_{Cc} (Y^{prioS^k_{Cc}} + Y^{prioF^k_{Cc}}) \qquad \text{(III.16)}$$

Ces fonctions de forçage combinées, permettent donc de regrouper les informations des différentes priorités, existantes sur l'ensemble des contraintes combinées.

La définition automatique des fonctions de forçage combinées, à partir des priorités structurelles et fonctionnelles, a été implémentée dans SEDMA (Annexe II-3.3).

Portail 6. ***Extraction des fonctions*** $F0c^k$ ***et*** $F1c^k$

La priorité fonctionnelle choisie pour cet exemple est rappelée ci-dessous :

« $Cc_1 = 1 \implies close = 0$ *» (priorité à open)*

Dans ces conditions, les fonctions de forçage combinées, pour l'exemple du portail sont présentées ci-dessous :

$$open(i.e.\ o_1) : \begin{cases} F0c^1 = F1s^2 & \text{extrait de } prioS^1_{Cc_1} \\ F1c^1 = 0 \end{cases}$$

$$close(i.e.\ o_2) : \begin{cases} F0c^2 = \overline{F1s^1} \cdot \overline{F1s^2} \cdot g_1 \cdot g_2 + F1s^1 & prioF^2_{Cc_1},\ prioS^2_{Cc_1} \\ F1c^2 = 0 \end{cases}$$

Rappel : g_1 *(resp.* g_2*) correspond à la valeur de la variable open (resp. close) à tester par le filtre.*

En s'appuyant sur un formalisme à base d'équations logiques, et sur l'exemple du portail,

cette section a montré qu'il est possible de modéliser le cahier des charges informel sous forme de contraintes logiques formelles. Cette modélisation s'appuie entre autres sur la définition de différentes fonctions, permettant la résolution d'une contrainte logique quelconque. Néanmoins, ces contraintes logiques doivent être vérifiées formellement sur leur ensemble afin de garantir leur fonctionnement. La section suivante (section III-4), propose de définir une propriété de cohérence ainsi qu'une approche de vérification formelle de cette propriété.

III-4 Vérification formelle d'un ensemble de contraintes logiques

III-4.1 Généralités

Une « proposition logique » peut ne pas avoir de solution, c'est-à-dire qu'il n'est pas toujours possible d'affecter une valeur à toutes les variables de la proposition et d'évaluer cette dernière à *vraie*. Ce problème fait partie de la logique propositionnelle en informatique et est connu sous le nom de *Problème de Satisfaction de Contraintes* (CSP). Différents types de CSP existent et dépendent du domaine des variables mises en jeu (entières, booléennes, etc.). Dans le cas où toutes les variables sont booléennes, le problème se nomme *problème de SATisfiabilité* (SAT) et est défini ci-dessous (Déf. 29).

Définition 29. *Une formule (ensemble de contraintes) est satisfiable s'il est possible de trouver une interprétation (i.e. affectation) qui rend la formule vraie.*

Dans le cas du filtre logique, les variables sont booléennes mais la propriété à vérifier est différente. En effet, le but n'est pas uniquement de savoir s'il existe une solution, mais de savoir s'il existera toujours un vecteur de sortie sûr à envoyer au système quelles que soient les valeurs des variables non-commandables.

Dans cette thèse, la notion de *cohérence* d'un ensemble de contraintes logiques est définie en deux parties. Tout d'abord l'existence d'une solution est vérifiée algébriquement (Déf.

30), puis la cohérence des contraintes et des priorités est vérifiée (Déf. 31).

Définition 30. *Un filtre logique est* ***cohérent algébriquement*** *s'il est possible de trouver une affectation des variables commandables qui valide l'ensemble des contraintes logiques quelles que soient les affectations des variables non-commandables.*

Hietter (2009) a démontré une condition nécessaire et suffisante pour l'existence d'une solution paramétrique à une équation de type $a \cdot \overline{x} + b \cdot x = 0$ appelée forme canonique (théorème 1). Ce théorème a également été étendu à plusieurs variables (Hietter, 2009).

Théorème 1. ***Extrait de Hietter (2009)***

Soient $\mathbb{B}$ *une algèbre de Boole et* a, b, $x \in \mathbb{B}$. *La solution de l'équation* $a \cdot \overline{x} + b \cdot x = 0$ *est* $x = a + \overline{b} \cdot p$ *avec* $p \in \mathbb{B}$ *un paramètre si et seulement si* $a \cdot b = 0$.

De plus, Hietter (2009) a également démontré qu'il était possible de transformer toute composition (produit logique) d'éléments d'un algèbre de Boole sous une forme canonique équivalente (Théorème 2 dans Hietter (2009)).

En utilisant les travaux de Hietter (2009), il est possible d'obtenir une condition nécessaire et suffisante à la cohérence algébrique des contraintes logiques (théorème 2).

Théorème 2. *La condition de cohérence de la synthèse algébrique est nécessaire et suffisante à la cohérence algébrique du filtre logique.*

Preuve :

En identifiant les variables commandables (resp. non-commandables) aux variables inconnues (resp. connues) de la synthèse algébrique, il est possible de définir les contraintes logiques comme des compositions de la synthèse algébrique.

Par conséquent, d'après les travaux de (Hietter, 2009), tout ensemble de contraintes logiques peut être transformé en une forme canonique équivalente.

Dès lors, la condition de cohérence de Hietter (théorème 1) est nécessaire et suffisante à l'existence d'une solution paramétrique.

De plus, une solution paramétrique signifie que, quelles que soient les valeurs de a, b et p (variables non-commandables), il existe une affectation de x (variable commandable) qui valide l'équation.

Donc, la condition de Hietter est nécessaire et suffisante à la cohérence algébrique du filtre logique.

Considérons l'ensemble des affectations valides et l'ensemble des affectations non-valides (Fig. 36), la cohérence algébrique signifie que l'ensemble des affectations valides est non vide.

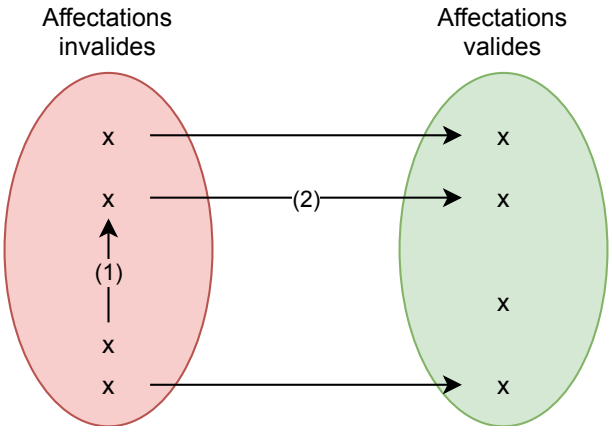


FIGURE 36 – Principe général de résolution

Néanmoins, cela ne garantit pas que, quelle que soit une affectation invalide donnée, une affectation valide puisse être calculée en appliquant l'algorithme de résolution. En effet,

l'application des fonctions de forçage ($F0s$, $F1s$, $F0c$ et $F1c$) permet de résoudre les contraintes violées par l'affectation actuelle, mais rien ne garantit que cette résolution ne va pas violer d'autres contraintes. L'algorithme de résolution doit alors être itératif : l'application des fonctions de forçage peut amener à devoir passer temporairement par différentes affectations invalides afin de pouvoir au final trouver une affectation valide. Ce mécanisme est illustré sur la figure 36 avec le passage (1) $\rightarrow$ (2). Afin de pouvoir garantir qu'il sera toujours possible de trouver une affectation valide lors de l'exécution en-ligne du filtre logique, il est nécessaire de définir une cohérence prenant en compte les priorités fonctionnelles. Dans ce cas, la condition de Hietter n'est plus suffisante, et la définition de la cohérence doit être enrichie (Déf. 31). En effet, les priorités fonctionnelles impliquent la prise en compte de séquentialité dans la résolution. Or, la vérification algébrique de Hietter ne s'adresse pas à ce problème.

Définition 31. *Un filtre logique est* ***cohérent*** *s'il est possible de trouver une affectation des variables commandables qui valide l'ensemble des contraintes quelles que soient les variables non-commandables en respectant les priorités fonctionnelles.*

Même si la condition de Hietter ne permet pas de garantir la cohérence du filtre logique, elle est tout de même nécessaire pour garantir que l'ensemble des solutions n'est pas vide.

Proposition 1. *La cohérence algébrique est une condition* ***nécessaire*** *à la cohérence du filtre logique.*

Exemple 2. *Soient a et b deux variables non-commandables. O_1, O_2 et O_3 trois variables commandables. Considérons les contraintes suivantes avec en* ***gras*** *les variables forcées via les priorités fonctionnelles :*

$$Cs_1 = a \cdot O_2; Cs_2 = \overline{a} \cdot b \cdot O_1$$

$$Cc_1 = O_1 \cdot \overline{\mathbf{O_2}}; Cc_2 = \mathbf{O_2} \cdot \overline{O_3}$$

Afin de vérifier la cohérence algébrique, l'outil dédié à la synthèse algébrique est utilisé : BESS[a]*. Seules les contraintes et la définition des variables sont à*

saisir (pas les priorités). L'outil vérifie la propriété de cohérence algébrique en appliquant le théorème 1 et indique la représentation paramétrique de l'espace de solution si la propriété est validée :

$$O_1 = p\_O_1 \cdot (\overline{a} \cdot \overline{b})$$
$$O_2 = \overline{a} \cdot \overline{b} \cdot p\_O_1 + p\_O_2 \cdot (\overline{a})$$
$$O_3 = (\overline{a} \cdot p\_O_2 + \overline{a} \cdot \overline{b} \cdot p\_O_1) + p\_O_3$$

Dans cet exemple le filtre logique est donc cohérent algébriquement car l'outil arrive à calculer une représentation paramétrique de l'espace des solutions. (note : p_O_1, p_O_2 et p_O_3 sont des paramètres Booléen quelconques à choisir) En considérant les priorités fonctionnelles et donc la résolution itérative qui en découle, il est possible de trouver une affectation des variables non-commandables et une pré-affectation des variables commandables à partir desquelles il ne sera pas possible de trouver une affectation valide des variables commandables :

$$ab|O_1O_2O_3$$
$$t_0 \;\; 10|110 \Rightarrow Cc_2 = 1 \Rightarrow O_2 := 0$$
$$t_1 \;\; 10|100 \Rightarrow Cc_1 = 1 \Rightarrow O_2 := 1$$
$$t_2 \;\; 10|110 \Rightarrow Cc_2 = 1 \Rightarrow O_2 := 0$$
$$\ldots$$

L'ensemble de contraintes est donc non-cohérent car un cycle de résolution apparaît. Néanmoins en changeant les priorités fonctionnelles, ce cycle disparaît et les contraintes deviennent cohérentes.

a. http ://www.lurpa.ens-cachan.fr/-226050.kjsp

Afin de vérifier cette propriété de cohérence, Riera *et al.* (2015) ont proposé différentes conditions **nécessaires** à la cohérence. Dans le cadre de cette thèse nous avons souhaité trouver une condition **nécessaire et suffisante** à la cohérence. Dans la suite de cette partie, la vérification de la cohérence, lorsque l'ensemble de contraintes n'est constitué que de contraintes simples, est tout d'abord présentée. Puis, une approche par exploration d'espace d'état est proposée, celle-ci fournit une condition nécessaire et suffisante à la cohérence d'un ensemble de contraintes logiques quelconques.

III-4.2 Cas particulier : cohérence d'un ensemble de contraintes simples

Lorsque l'ensemble de contraintes est constitué uniquement de contraintes simples, la vérification de la cohérence peut être résolue algébriquement. La définition 32 définit ce cas particulier.

Définition 32. *L'ensemble des contraintes simples est cohérent si et seulement si quelle que soit la valeur des variables non commandables, il existe au moins une affectation des variables commandables qui vérifie l'ensemble des contraintes simples.*

Le théorème 3 permet de vérifier la cohérence d'un ensemble de contraintes simples.

Théorème 3. *La propriété suivante est nécessaire et suffisante à la cohérence des contraintes simples :*

$$\forall k \in \{1, \ldots, N_o\}, F0s^k \cdot F1s^k = 0$$

Preuve :

Par définition (section III-3.3.1), les contraintes simples portant sur la variable $o_k \in O$ sont résolues lorsque l'équation III.10 est vérifiée. (rappel de l'équation : $F0s^k \cdot o_k + F1s^k \cdot \overline{o_k} = 0$)

Or cette équation respecte les conditions du théorème de Hietter (Theorème 1). Donc, d'après le théorème de Hietter, l'équation $F0s^k \cdot F1s^k = 0$ est nécessaire et suffisante à l'existence d'une solution.

Il est néanmoins à noter que, comme montré dans Riera *et al.* (2015), cette propriété n'est pas nécessaire et suffisante à la cohérence d'un ensemble de contraintes logiques dans le cas général (contraintes simples et combinées avec priorités). Dans la section suivante (section III-4.3), une approche est proposée permettant l'obtention d'une condition nécessaire et suffisante, à la cohérence d'un ensemble de contraintes logiques dans le cas général.

III-4.3 Proposition d'une approche de vérification formelle de la cohérence

L'approche proposée permet l'obtention d'une condition nécessaire et suffisante à la cohérence d'un ensemble de contraintes logiques (Fig. 37). Dans cette approche de vérification, un **problème** correspond à un ensemble de contraintes logiques avec les priorités fonctionnelles associées.

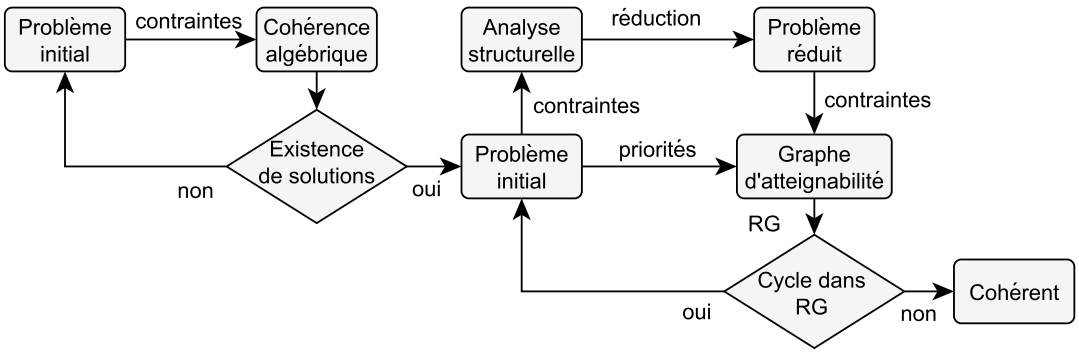


FIGURE 37 – Méthode de vérification formelle de la cohérence

L'approche débute par une vérification de la cohérence algébrique en considérant les contraintes sans les priorités fonctionnelles. Celle-ci est effectuée à l'aide la synthèse algébrique (Hietter, 2009) et permet de vérifier l'**existence** d'un espace de solution (section III-4.3.1). Si au moins une solution existe, alors l'analyse peut continuer et se déroule en deux étapes. Premièrement, une **analyse structurelle** des contraintes est effectuée (section III-4.3.2), permettant de réduire le nombre de contraintes à vérifier. Basé sur ce problème réduit, un **graphe d'atteignabilité** (RG) est construit (section III-4.3.3). Ce dernier, permet de mettre en évidence une **condition nécessaire et suffisante** à la cohérence d'un ensemble de contraintes logiques quelconques (section III-4).

Cette approche de vérification formelle de la cohérence a été implémenté dans le logiciel SEDMA (Annexe II-3.3). Le fichier d'entrée présenté dans l'annexe, contenant les contraintes logiques et les priorités fonctionnelles, définit le problème initial. La vérification de l'existence d'une solution, l'analyse structurelle, la réduction du problème, la génération et l'analyse des différents graphes sont automatisés.

III-4.3.1 Existence d'un espace de solutions

Dans un premier temps, la synthèse algébrique est utilisée afin de vérifier l'existence d'un espace de solution.

Pour ce faire, l'outil dédié à la synthèse algébrique est utilisé : BESS [1]. Cet outil permet de vérifier algébriquement et efficacement l'existence d'une solution à partir d'une description des contraintes et des variables.

Si une solution paramétrique existe, la propriété de cohérence algébrique est vérifiée. Dans le cas contraire, BESS fournit un contre-exemple sous la forme d'une équation logique. L'analyse de ce contre-exemple peut permettre d'identifier des erreurs durant la conception des contraintes logiques.

III-4.3.2 Réduction du problème par analyse structurelle

Dans le but de réduire le nombre de contraintes à vérifier, les interactions structurelles entre les contraintes sont représentées à l'aide d'un graphe non orienté. Ce graphe est appelé Graphe Structurel (SG) et est défini ci-dessous (Déf. 33).

Définition 33. *SG = (C, E) tel que,*

- *C est un ensemble de nœuds. Chaque nœud représente une contrainte simple ou combinée.*
- *E est un ensemble d'arêtes.*

Une arête entre deux contraintes signifie que la résolution de l'une peut violer la seconde. L'existence d'une arête entre deux contraintes est définie à l'aide de deux conditions (Déf. 34).

Définition 34. *Soient deux contraintes C_i et C_j (Déf. 17), une arête existe entre les nœuds représentant ces contraintes si et seulement si :*

1- $Y^{C_i} \cdot Y^{C_j} \neq 0$;

ET

2- $O^{C_i} \cdot O^{C_j} = 0$.

1. http ://www.lurpa.ens-cachan.fr/-226050.kjsp

En d'autres termes, une arête existe entre deux contraintes uniquement si :

1- leurs parties non-commandables respectives peuvent être *vraies* simultanément ;

2- il existe au moins une variable commandable sous forme complémentée dans l'une des contraintes et non complémentée dans l'autre.

En considérant les contraintes présentées dans l'exemple 2 (rappelées ci-dessous), le graphe structurel associé est présenté dans la figure 38.

$$Cs_1 = a \cdot O_2 \qquad Cs_2 = \overline{a} \cdot b \cdot O_1$$
$$Cc_1 = O_1 \cdot \overline{O2} \qquad Cc_2 = O_2 \cdot \overline{O_3}$$

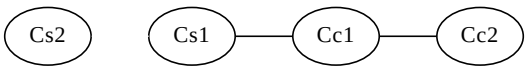


FIGURE 38 – Graphe structurel pour l'exemple 2

Le graphe structurel permet de connaître rapidement les liens de résolution potentiels entre contraintes. Il est à noter que seules les contraintes sont nécessaires à la construction de SG, en effet les priorités fonctionnelles ne sont pas utilisées.

Étant donné les conditions d'existence d'une arête, certains nœuds (et donc contraintes) peuvent se retrouver isolés (aucun arc n'est lié). Considérant le problème de l'analyse de la cohérence, ces contraintes peuvent alors être ignorées. En effet, la résolution d'une contrainte isolée ne violera jamais d'autres contraintes et vice versa. De plus, la non prise en compte de ces contraintes permet d'ignorer totalement certaines variables (celles n'apparaissant que dans des contraintes isolées).

Les contraintes et variables restantes après réduction permettent de construire un problème *réduit* du problème *initial* (exemple 3). Basé sur ce problème réduit, l'étape d'analyse par graphe d'atteignabilité peut être effectuée.

Exemple 3. *A partir du problème initial, le graphe structurel est construit (Fig. 38), puis les contraintes isolées sont supprimées pour former un nouvel ensemble de contraintes :*

$$Cs_1 = a \cdot O_2$$
$$Cc_1 = O_1 \cdot \overline{O2} \qquad Cc_2 = O_2 \cdot \overline{O_3}$$

III-4.3.3 Définition du graphe d'atteignabilité

Après avoir réduit le problème à vérifier, le but est de construire un graphe permettant la vérification de la cohérence. Ce graphe doit contenir l'ensemble des chemins de résolution possibles, c'est-à-dire comment, à partir d'une affectation non-valide, il est possible d'atteindre un état valide (Fig. 36). Si pour toutes les affectations non-valides il est possible d'atteindre un état valide, alors le problème est cohérent par définition.

Le graphe d'atteignabilité (RG) est un graphe orienté, défini ci-dessous (Déf. 35).

Définition 35. *RG = (V, A) tel que,*

- *V est l'ensemble des nœuds. dim(V) = 2^n avec n le nombre de variables commandables et non-commandables. Chaque nœud est une affectation complète des variables non-commandables et des variables commandables. Chaque nœud est labellisé par les contraintes violées par cette affectation (s'il y en a).*
- *A est un ensemble d'arcs. Un arc représente comment une affectation de départ est modifiée (variables forcées) pour résoudre les contraintes violées par cette affectation.*

A partir d'un problème (contraintes et priorités), qu'il soit réduit ou non, il est possible de construire le graphe d'atteignabilité en 4 étapes :

1- Calculer les affectations valides.
2- Calculer les affectations non-valides.
3- Labelliser les affectations non-valides par les contraintes qu'elles violent.
4- Relier les affectations entre elles par des arcs à l'aide des priorités.

Le calcul des affectations valides est possible avec un solveur SAT (Du *et al.*, 1997). Étant donné un ensemble de clauses (équations Booléennes), un solveur SAT est capable de fournir l'ensemble des affectations des variables qui valide l'ensemble des clauses.

Chaque affectation qui n'est pas listée par le solveur SAT est obligatoirement non-valide. Pour chacune d'entre elles, il est nécessaire de les analyser pour savoir quelles sont les contraintes violées. Le nœud représentant l'affectation non-valide est ensuite labellisé par

les contraintes violées.

Pour chaque affectation non-valide, et considérant les contraintes violées par celle-ci, les règles de résolution associées à ces contraintes sont appliquées. Comme vu précédemment, la résolution d'une contrainte est déterministe. Donc, il n'existe qu'un seul arc sortant d'un nœud. Si pour une affectation, deux contraintes sont en contradiction (l'une force une variable à *vrai* et l'autre à $faux$) l'arc est changé par une boucle sur le même nœud.

Certaines affectations valides peuvent se retrouver sans arc entrant, dans ce cas les nœuds correspondants sont supprimés du graphe final pour réduire sa taille. Il est également à noter qu'il n'existe pas d'arc sortant d'un nœud valide puisque aucune contrainte n'est à résoudre à partir de ces nœuds.

Chaque nœud du graphe d'atteignabilité représente une affectation possible des variables avant la résolution du filtre logique. Chaque nœud peut être initial, de plus étant donné un nœud initial, l'affectation finale sera toujours la même car la procédure de résolution est déterministe.

Exemple 4. *Considérons les contraintes de l'exemple 3, et les priorités fonctionnelles suivantes :* $Cc_1 = 1 \Longrightarrow O_1 := 0$ *et* $Cc_2 = 1 \Longrightarrow O_2 := 0$. *Avec ces priorités, le graphe d'atteignabilité résultant est présenté sur la figure 39.*

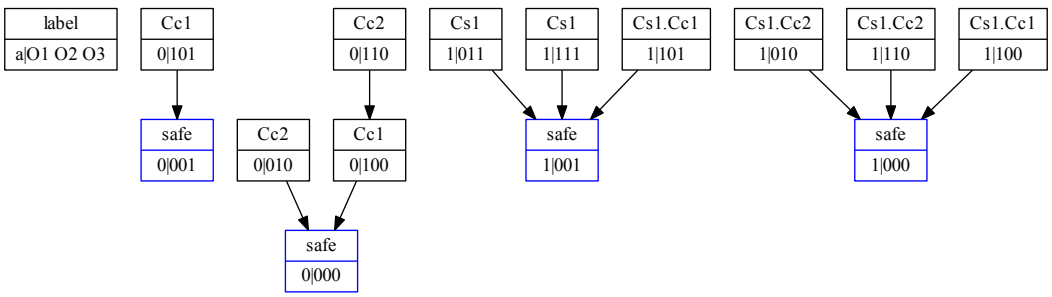


FIGURE 39 – Graphe d'atteignabilité pour les priorités fonctionnelles de l'exemple 4

III-4.3.4 Analyse du graphe d'atteignabilité

En se basant sur la définition et la construction du graphe d'atteignabilité, la propriété de cohérence (Déf. 31) est équivalente à une propriété d'atteignabilité dans RG (Prop. 2).

Proposition 2. *Le problème est cohérent si et seulement si à partir de toute affectation non-valide, une affectation valide est atteignable dans le graphe d'atteignabilité.*

Il est à noter que cette propriété nécessite l'exploration complète du graphe, ce qui s'avère inefficace dans la plupart des cas. En effet n variables implique 2^n états à explorer, le risque d'explosion combinatoire est donc élevé dans ce cas. Néanmoins, considérant la construction de RG, il n'y a qu'un seul arc sortant de chaque nœud. Il est donc possible de vérifier qu'il n'existe pas de cycle dans RG (Prop. 3). En effet, si un cycle existe, cela signifie qu'il existe au moins une affectation non-valide qui ne peut pas être changée en une affectation valide.

Proposition 3. *Le problème est non-cohérent si et seulement si il existe au moins un cycle dans RG.*

Dans la plupart des cas, la proposition 3 est plus facile à vérifier que la proposition 2. En effet, la vérification d'une propriété en « il existe » est en général moins coûteuse en temps de calcul qu'une propriété en « pour tout ».

Exemple 5. *Si l'on considère les priorités fonctionnelles présentées dans l'exemple 4, ainsi que le graphe d'atteignabilité correspondant (Fig. 39), alors le problème (contraintes et priorités fonctionnelles) est cohérent. En effet, il n'existe pas de cycle.*
A présent, considérons les priorités fonctionnelles suivantes : $Cc_1 = 1 \Longrightarrow O_2 := 1$ *et* $Cc_2 = 1 \Longrightarrow O_2 := 0$. *Le graphe d'atteignabilité correspondant est présenté figure 40.*
Il existe un cycle entre $Cc1$ *et* $Cc2$, *donc il existe au moins une affectation initiale ne permettant pas d'atteindre un état respectant toutes les contraintes. Par conséquent, ce problème est non cohérent. De plus, l'incohérence est due aux priorités fonctionnelles car les mêmes contraintes sont cohérentes avec d'autres priorités (Fig. 39).*

Avec cette approche par graphe, il est donc possible de vérifier la cohérence d'un ensemble

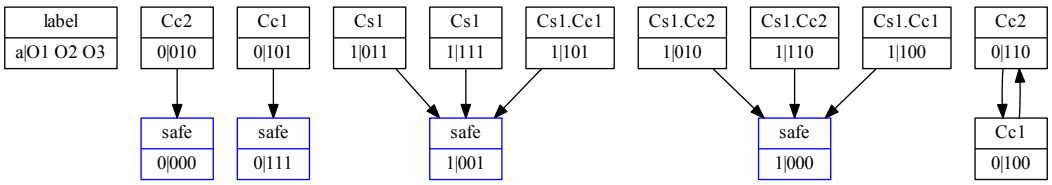


FIGURE 40 – Graphe d'atteignabilité pour les priorités fonctionnelles de l'exemple 5

de contraintes logiques. De plus, il est possible de faire remonter le(s) cycle(s) à l'utilisateur lors de la phase de définition des contraintes, cela permet d'identifier plus facilement les problèmes entre les contraintes.

Avant de conclure ce chapitre III, il est possible de montrer des liens entre la théorie de la commande par supervision (SCT) et le filtre logique.

III-5 Apports mutuels entre SCT et filtre logique : exemple du chat et de la souris

Cette section a pour objectif de montrer que le filtre logique permet l'implémentation efficace du superviseur calculé par la SCT. Pour cela, une extension de la SCT permettant la transformation d'un superviseur en équations logiques est utilisée (Miremadi *et al.*, 2011).

Cette étude permet également de montrer que la SCT peut servir dans l'approche par filtre logique, afin de générer automatiquement l'ensemble des contraintes définissant le filtre logique développé dans cette thèse. Ce dernier point implique une modification de la méthodologie d'obtention du filtre logique (Fig. 41).

Dans un premier temps, la section III-5.1 présente comment il est possible, à partir de la modélisation proposée dans la section II-3.3, de générer un ensemble de contraintes logiques traduisant le superviseur de la SCT (Miremadi *et al.*, 2011). Puis, la section III-5.2 utilise cet ensemble de contraintes comme entrée de l'approche par filtre logique. Ces contraintes sont ensuite analysées et comparées à un ensemble défini manuellement.

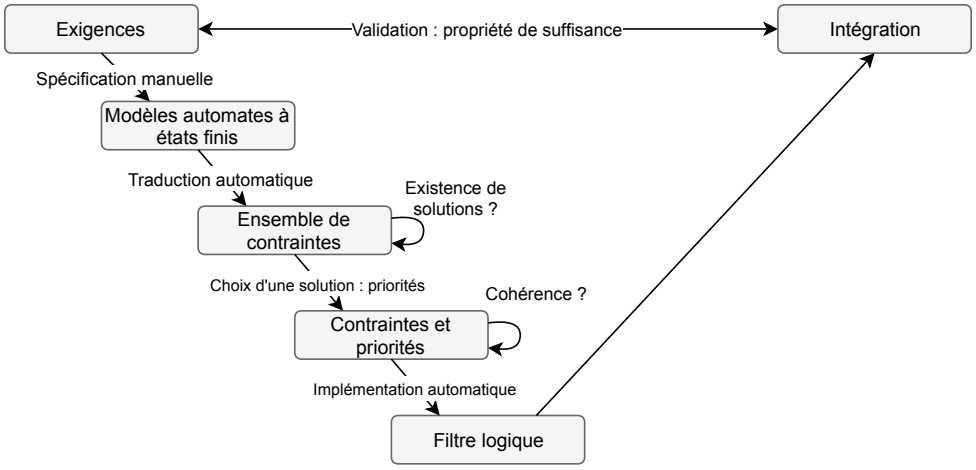


FIGURE 41 – Variante méthodologique pour l'obtention du filtre utilisant la SCT

III-5.1 Génération automatique des contraintes logiques

La méthodologie développée dans Miremadi *et al.* (2011) utilise les arbres de décisions binaires (BDD : Binary Decision Diagram) afin de représenter les modèles. Les BDD permettent des calculs efficaces même sur des modèles de grande dimension.

Le principe de calcul de Miremadi *et al.* (2011) peut être résumé en plusieurs étapes :

1- L'automate $S_0 = G||K$ est calculé par composition, avec G le modèle du système et K le modèle des spécifications.

2- Certains états doivent être exclus dans le but d'obtenir le superviseur. Trois types d'états sont alors définis : les états interdits, les états autorisés et les états quelconques. Ces états sont identifiés dans S_0.

3- Un ensemble de contraintes logiques est calculé, chaque contrainte autorise ou interdit un événement à apparaître afin de ne jamais pouvoir atteindre un état interdit à partir d'un état quelconque ou autorisé de S_0.

4- Les contraintes sont réduites automatiquement afin de minimiser le nombre de variables apparaissant dans celles-ci.

Avec cette approche, le superviseur final n'est pas décrit à l'aide d'un automate à états finis, mais avec un ensemble de contraintes logiques. Une contrainte peut être assimilée à la suppression d'une transition dans un automate à états finis.

Les différents algorithmes et calculs de cette méthode sont implémentés dans le logiciel

Supremica (Akesson *et al.*, 2006). Cet outil est accessible lorsque Supremica est en mode « editor » puis dans l'onglet « Analyze/Symbolic (BDD) Synthesis/Optimization on TEFAs... ». La fenêtre qui s'ouvre alors a besoin d'être configurée comme présentée sur la figure 42.

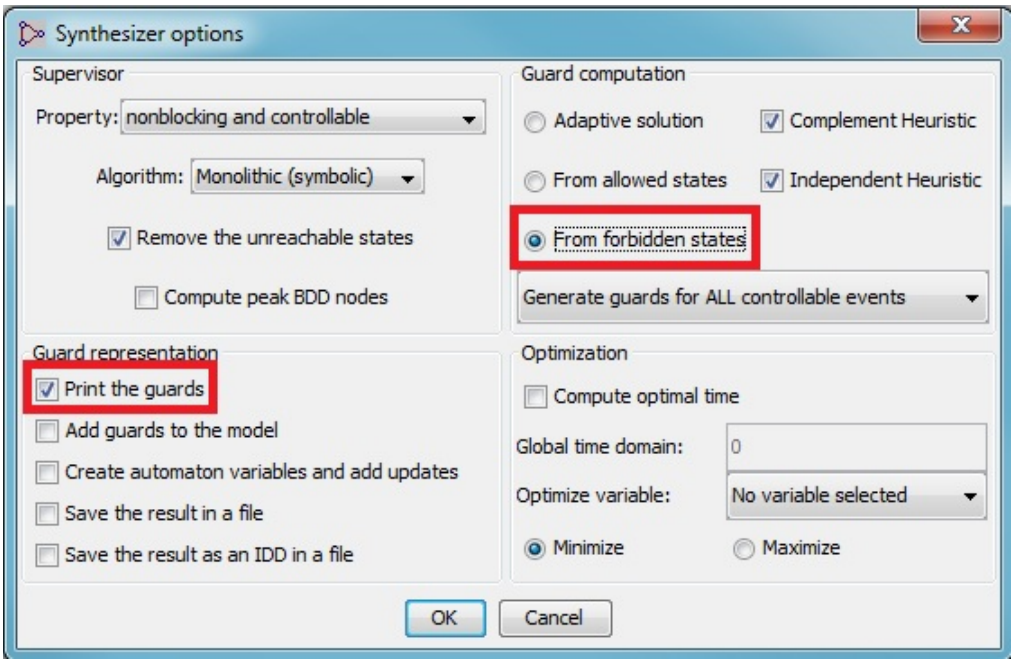


FIGURE 42 – Configuration du générateur de gardes

L'option « Print the guards » permet l'affichage du résultat dans la console de Supremica. L'option « From forbidden states » permet un formatage du résultat facile à utiliser comme présenté par la suite. Par défaut, l'outil va générer une garde pour chaque événement contrôlable, il est néanmoins possible de n'en sélectionner qu'un seul.

Plusieurs types de résultats sont alors possibles :

1- FORBIDDEN guard for event e : This event is always ENABLED by the supervisor.

$\longrightarrow$ cet événement peut toujours être autorisé

2- FORBIDDEN guard for event e : This event is always DISABLED by the supervisor.

$\longrightarrow$ cet événement ne doit jamais être autorisé

3- FORBIDDEN guard for event e : This event is always DISABLED by the supervisor (Blocked in the synchronization process).

$\longrightarrow$ cet événement ne doit jamais être autorisé, néanmoins il ne peut jamais apparaître si l'état courant est correct.

4- FORBIDDEN guard for event e : *condition*

$\longrightarrow$ Si la *condition* est validée, l'événement e est autorisé

A partir de ces résultats il est possible d'extraire les contraintes logiques immédiatement. La solution 1 signifie qu'aucune contrainte ne portera sur cette variable. La solution 2 indique que la variable doit toujours être forcée à 0. La solution 3 indique que la variable doit également toujours être forcée à 0 mais que normalement il n'est pas nécessaire de le faire car en fonctionnement normal cela n'arrivera jamais. La solution 4 est la plus intéressante, en effet la contrainte résultante est : $\overline{condition} \implies \overline{e} \Leftrightarrow \overline{condition}.e = 0$. Prenons un exemple tiré du chat et de la souris, la porte $C2$ ne doit pas être ouverte quand la souris est dans la pièce 2 ($mouse\_curr = 2$) :

FORBIDDEN guard for event $C2$: $mouse\_curr \neq 2$
$\Leftrightarrow (\overline{mouse\_curr \neq 2}) \cdot C2 = 0$
$\Leftrightarrow (mouse\_curr = 2) \cdot C2 = 0$
$\Leftrightarrow r2m \cdot C2 = 0$

mouse_curr est une variable entière interne à Supremica permettant de connaître l'état courant du modèle « mouse » présenté précédemment. Dans notre cas nous avons directement une variable logique indiquant si le chat est dans la pièce 2 : $r2m$, c'est pourquoi la contrainte résultante est : $r2m \cdot C2 = 0$.

Si la condition comporte des ET logique (noté « & » dans Supremica) il est nécessaire de créer 2 contraintes :

FORBIDDEN guard for event $C1$: $((mouse\_curr \neq 1)$ & $(mouse\_curr \neq 3))$
$\Leftrightarrow \overline{((mouse\_curr \neq 1) \cdot (mouse\_curr \neq 3))} \cdot C1 = 0$
$\Leftrightarrow (r1m + r3m) \cdot C1 = 0$
$\Leftrightarrow (r1m \cdot C1) + (r3m \cdot C1) = 0$

Enfin, lorsque l'événement est un événement « simultané » comme présenté précédemment, cela va se traduire par une contrainte combinée :

FORBIDDEN guard for event $M5\_C1$: This event is always DISABLED by the supervisor.
$\Leftrightarrow M5 \cdot C1 = 0$

En effectuant les mêmes transformations pour tous les événements commandables de l'exemple, l'ensemble de contraintes généré automatiquement est présenté dans le tableau 13. Les variables en majuscules sont commandables et celles en minuscules sont non-commandables.

TABLE 13 – Contraintes générées automatiquement

Contraintes simples		Contraintes combinées	
$r2c \cdot M1 = false$	$r1m \cdot C1 = false$	$M1 \cdot C2 = false$	$M6 \cdot C3 = false$
$r1c \cdot M2 = false$	$r3m \cdot C1 = false$	$M4 \cdot C5 = false$	$M3 \cdot C3 = false$
$r3c \cdot M2 = false$	$r2m \cdot C2 = false$	$M5 \cdot C1 = false$	
$r0c \cdot M3 = false$	$r0m \cdot C3 = false$	$M5 \cdot C4 = false$	
$r4c \cdot M4 = false$	$r1m \cdot C4 = false$	$M2 \cdot C1 = false$	
$r3c \cdot M5 = false$	$r3m \cdot C4 = false$	$M2 \cdot C4 = false$	
$r1c \cdot M5 = false$	$r4m \cdot C5 = false$	$M6 \cdot C6 = false$	
$r0c \cdot M6 = false$	$r0m \cdot C6 = false$	$M3 \cdot C6 = false$	

Bien que le problème comporte plusieurs dizaines de variables, chaque contrainte est assez simple à lire. Les contraintes simples de cet exemple sont toutes de la même forme : si un animal est dans une pièce, alors une porte est fermée. Prenons l'exemple de la première : $r2c.M1 = false$, $r2c$ signifie que le chat est dans la pièce 2, donc la porte M1 doit être fermée ($M1 = false$) afin de garantir que la souris ne puisse pas entrer.

Les contraintes combinées ont elles aussi toutes la même forme : elles empêchent le mouvement simultané des deux animaux si ce mouvement entraine obligatoirement une situation dangereuse. Toutes les combinaisons de portes ne sont pas interdites. Par conséquent, il existe des déplacements simultanés qui ne posent pas de problème.

III-5.2 Analyse de l'ensemble de contraintes généré automatiquement

Bien que cet ensemble de contraintes soit généré automatiquement à l'aide de la SCT, il est important de l'analyser. La première analyse à effectuer pour tester un ensemble de contraintes, est de vérifier sa cohérence comme présentée dans la section III-4.3.

L'application de l'approche d'analyse graphique de la cohérence conclut rapidement que l'ensemble généré automatiquement est **cohérent**. En effet, dans le graphe structurel (représentant les liens entre les contraintes), aucun arc n'existe. Le lecteur peut le vérifier

rapidement en analysant les contraintes, en effet il n'existe aucune variable sous forme complémentée, donc la mise à 0 d'une variable pour la résolution d'une contrainte n'entrainera jamais la violation d'une autre contrainte.

La seconde étape d'analyse est de vérifier la suffisance de l'ensemble de contraintes, c'est-à-dire, est-ce que cet ensemble permet de garantir qu'aucun état interdit ne soit atteignable. Pour cela, l'approche basée sur du model-checking développée dans Marangé (2008) a été utilisée. La propriété à vérifier est assez simple à exprimer. En effet, elle consiste à vérifier que des couples d'états interdits ne sont jamais atteignables : chats et souris dans la même pièce (5 états), chat en 1 et souris en 3 (1 état) et chat en 3 et souris en 1 (1 état). L'application de l'approche d'analyse de la suffisance indique que l'ensemble généré automatiquement est **suffisant** pour garantir la survie de la souris.

L'ensemble généré automatiquement, à l'aide de l'approche de modélisation proposée et du logiciel Supremica, est donc cohérent et suffisant pour cet exemple du chat et de la souris. Malgré tout, cet ensemble généré automatiquement est-il le « meilleur » possible ?

Pour répondre à cette question, il est important de signaler qu'une caractéristique de l'approche par filtre logique est de définir un filtre qui soit le plus permissif possible. En d'autres termes, nous souhaitons que l'espace d'états interdits par le filtre soit le plus petit possible. Or, il est possible de trouver un autre ensemble de contraintes, **cohérent** et **suffisant**, qui soit plus permissif que le précédent.

L'ensemble de contraintes suivant (Tab. 14) a été créé manuellement en suivant des règles simples :

- Pour chaque porte : identifier la ou les situations dangereuses (couple de positions) imposant la fermeture de la porte. Cette règle permet l'obtention de toutes les contraintes simples.
- Pour chaque couple de positions : identifier si un mouvement simultané des deux animaux amène à une situation interdite, si oui alors interdire l'ouverture des deux portes simultanément. Cette règle permet l'obtention de toutes les contraintes combinées.

Afin de comparer ces deux ensembles de contraintes, le nombre de vecteurs autorisés par chaque ensemble doit être calculé. Pour cela, un solveur SAT (Du *et al.*, 1997) a été utilisé. Le solveur SAT va chercher tous les vecteurs qui sont solutions de l'ensemble des

TABLE 14 – Contraintes générées manuellement

Contraintes simples		Contraintes combinées
$r2c \cdot r0m \cdot M1 = false$	$r0c \cdot r1m \cdot C1 = false$	$r1c \cdot r0m \cdot M1 \cdot C2 = false$
$r1c \cdot r2m \cdot M2 = false$	$r0c \cdot r3m \cdot C1 = false$	$r3c \cdot r0m \cdot M4 \cdot C5 = false$
$r3c \cdot r2m \cdot M2 = false$	$r1c \cdot r2m \cdot C2 = false$	$r0c \cdot r4m \cdot M5 \cdot C1 = false$
$r0c \cdot r1m \cdot M3 = false$	$r2c \cdot r0m \cdot C3 = false$	$r0c \cdot r4m \cdot M5 \cdot C4 = false$
$r4c \cdot r0m \cdot M4 = false$	$r0c \cdot r1m \cdot C4 = false$	$r0c \cdot r2m \cdot M2 \cdot C1 = false$
$r3c \cdot r4m \cdot M5 = false$	$r0c \cdot r3m \cdot C4 = false$	$r0c \cdot r2m \cdot M2 \cdot C4 = false$
$r1c \cdot r4m \cdot M5 = false$	$r3c \cdot r4m \cdot C5 = false$	$r4c \cdot r3m \cdot M6 \cdot C6 = false$
$r0c \cdot r3m \cdot M6 = false$	$r4c \cdot r0m \cdot C6 = false$	$r4c \cdot r1m \cdot M3 \cdot C6 = false$
		$r2c \cdot r3m \cdot M6 \cdot C3 = false$
		$r2c \cdot r1m \cdot M3 \cdot C3 = false$

contraintes. Les résultats sont résumés dans le tableau 15.

TABLE 15 – Analyse des ensembles par solveur SAT

	Contraintes automatiques	Contraintes manuelles
Nombre de variables	23	23
Nombre total de vecteurs	8388608	8388608
Nombre de vecteurs valides	40960	944425
Nombre de vecteurs non-valides	8347648	7444183

L'ensemble créé manuellement autorise environ 20 fois plus de vecteurs que l'ensemble généré automatiquement. Ceci est assez étonnant car l'approche de génération automatique proposée, est basée sur la SCT. Or, la SCT a pour objectif de fournir un superviseur qui soit le plus permissif possible. Nous suspectons que ce phénomène soit lié à l'étape de réduction effectuée par Supremica lors de la traduction du superviseur sous forme d'ensemble de contraintes (Miremadi et Voronov, 2012). En effet, plus la contrainte est grande (contient un grand nombre de variables), plus elle peut être considérée comme permissive (exemple 6).

Exemple 6. *Considérons 3 variables logiques* $\{a, b, c\}$ *ainsi que les contraintes* $c_1 : a \cdot b = 0$ *et* $c_2 : a \cdot b \cdot c = 0$, c_1 *interdit deux vecteurs alors que* c_2 *n'en interdit qu'un seul.*

A ce jour, ce problème de minimisation de l'ensemble de contraintes reste ouvert et n'a pas été traité dans cette thèse.

III-5.3 Discussion

Comme montré précédemment (section III-5.2), l'ensemble de contraintes généré automatiquement sur l'exemple du chat et de la souris est cohérent et suffisant. Néanmoins l'étude générale de ces deux propriétés, lorsque l'on considère un ensemble généré automatiquement avec l'approche symbolique de Supremica, reste un problème ouvert. Deux éléments de réponse peuvent être avancés :

- La suffisance doit pouvoir être garantie en considérant l'approche de synthèse de la SCT. En effet, l'ensemble de contraintes est une traduction symbolique du superviseur de la SCT. Or, ce superviseur garantit qu'aucun état interdit (défini par les modèles de spécifications), ne soit accessible par le système contrôlé. La non-accessibilité des états interdits doit alors pouvoir être rapprochée de la notion de suffisance du filtre logique.
- En suivant l'approche de modélisation présentée dans cette section, ainsi que l'algorithme de génération des contraintes présenté dans (Miremadi et Voronov, 2012), la cohérence de l'ensemble généré semble être garantie par construction. En effet, les contraintes générées ne devraient jamais contenir de variables complémentées ($\overline{M1}$ par exemple). Donc en se basant sur l'approche d'analyse de la cohérence présentée dans la section III-4.3, le graphe structurel ne contiendrait jamais d'arcs et donc l'ensemble serait toujours cohérent par construction.

Comme discuté précédemment dans la section II-3, la synchronisation des événements n'est pas une opération anodine. Il peut sembler logique que l'événement résultant de la synchronisation de deux événements commandables soit également contrôlable. De même pour deux événements non-commandables, l'événement résultant est non-contrôlable. Néanmoins dans le cas où les deux événements à synchroniser n'ont pas la même propriété de contrôlabilité, cette notion de synchronisation est rapidement évoquée dans la Section 4.10 de Wonham (2015). Néanmoins, à notre connaissance, la question de la synchronisation d'événements reste ouverte.

III-6 Conclusion

Ce chapitre a présenté différentes contributions relatives à la définition et la conception d'un filtre logique.

Dans un premier temps, la section III-2 a présentée les étapes de développement d'un filtre logique. Puis, une organisation de celles-ci a été proposée sous forme de cycle en V formel.

Dans la section III-3, une formalisation des contraintes logiques a été proposée en se basant sur la logique Booléenne et sur la notion de littéraux logiques. Par la suite, la notion de priorité, introduite dans des travaux précédents (Coupat, 2014), a également été formalisée et généralisée à n'importe quels types de contrainte logique. A l'aide de cette nouvelle formalisation des contraintes et des priorités, la résolution d'une contrainte logique violée a également été généralisée.

En se basant sur cette nouvelle formalisation, la section III-4 a proposée une approche de vérification formelle de la cohérence d'un ensemble de contraintes logiques. Cette propriété de cohérence a été introduite précédemment (Riera *et al.*, 2015), et des conditions nécessaires ont été proposées. Dans cette section III-4, une condition nécessaire et suffisante a été proposée en se basant sur une construction et une analyse d'un graphe d'atteignabilité.

Enfin, la section III-5 a proposé une approche permettant, à partir de modèles automates à états finis et de la SCT, de générer automatiquement un ensemble de contraintes logiques. Cette approche a l'avantage de ne pas nécessiter l'écriture manuelle des contraintes logiques, mais nécessite tout de même la modélisation du système et des exigences sous formes d'automates à états finis. Cette section III-5 a également montré qu'un filtre logique, peut permettre d'implémenter efficacement un superviseur au sens de la SCT.

Pour conclure, ce chapitre a proposé différentes contributions permettant la définition d'un filtre logique de manière formelle (définitions, résolution, vérification). Le chapitre suivant se focalise sur l'étape suivante du cycle de développement proposé : l'implémentation.

CHAPITRE IV

Algorithmes et outils relatifs à l'approche par filtre logique

IV-1 Introduction

Dans le chapitre précédent, les contraintes logiques ont été introduites et définies. Le problème de la cohérence d'un ensemble de contraintes logiques a également été présenté, et une approche de vérification de cette cohérence a été proposée.

Les aspects théoriques du chapitre III permettent de répondre aux besoins des phases de formalisation, de conception et de vérification formelle de l'approche cyclique de conception proposée. A l'issue de ces phases, l'ensemble de contraintes a été vérifié formellement pour garantir la cohérence.

Ce chapitre IV se focalise à présent sur l'étape suivante du cycle de développement : l'implémentation du filtre logique (Fig. 43). Le but est de proposer des algorithmes, implémentables dans un automate programmable industriel, permettant de trouver en-ligne une affectation des sorties respectant toutes les contraintes. De plus, ces algorithmes doivent être déterministes, c'est-à-dire qu'à un vecteur d'entrée donné, le vecteur de sortie doit toujours être le même.

Dans un premier temps, la section IV-2 présente un algorithme itératif simple à implémenter, basé sur l'algorithme présenté dans Coupat (2014). Par la suite, dans la section IV-3, nous proposons deux algorithmes basés sur des techniques issues de la communauté

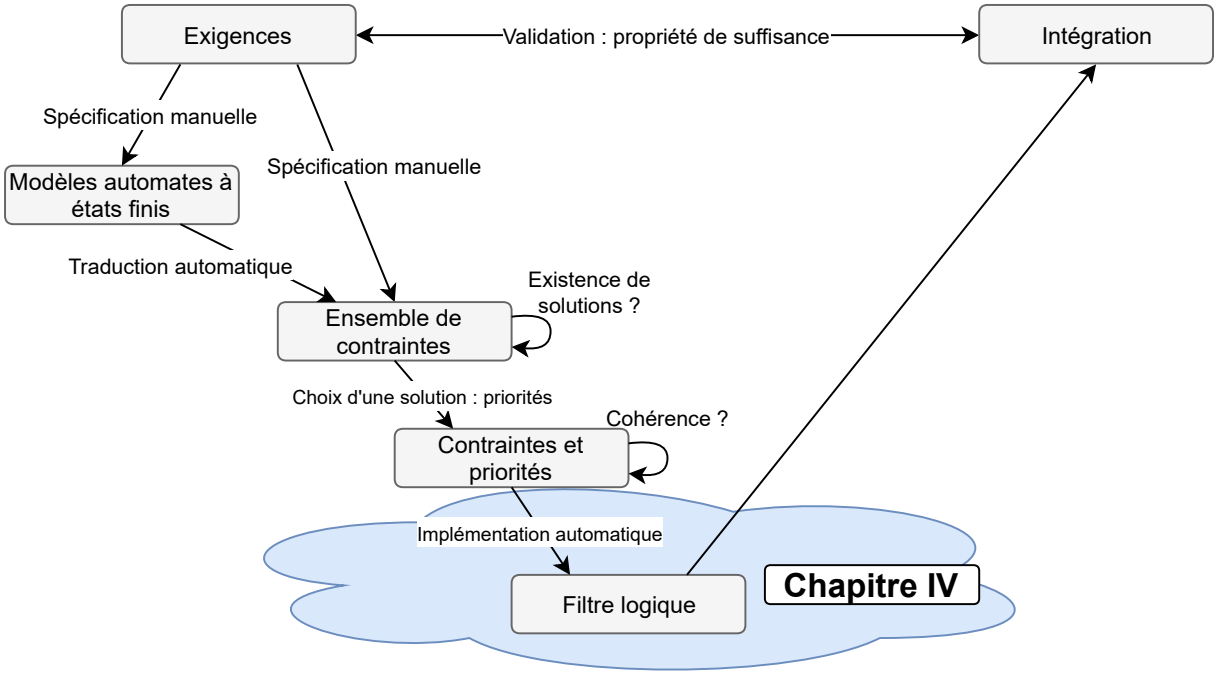


FIGURE 43 – Étapes du cycle de développement couvertes par le chapitre IV

informatique : les solveurs SAT. Ces propositions sont ensuite discutées et comparées dans la section IV-4.

IV-2 Algorithme itératif

Pour rappel, dans les travaux précédents (Coupat, 2014), la notion de priorité entre les variables d'une contrainte combinée est définie en utilisant des matrices de priorité. L'algorithme utilise donc cette structure matricielle, néanmoins cela limite l'expressivité des contraintes combinées. En effet, celles-ci ne peuvent être composées que de deux variables commandables, dans le cas contraire l'algorithme ne fonctionne pas. De plus, étant donné que la cohérence des contraintes ne peut pas être garantie, l'algorithme intègre un mécanisme de sécurité permettant d'éviter un *deadlock*, dans le cas où aucune solution n'existe.

Dans cette thèse, nous avons proposé une nouvelle modélisation des priorités, celle-ci fonctionne pour n'importe quelle taille de contrainte combinée (section III-3.3.2). Par ailleurs, nous avons montré qu'il est possible de garantir formellement la cohérence d'un ensemble de contraintes. L'algorithme proposé dans cette thèse (Fig. 44) est donc une amélioration de celui proposé dans Coupat (2014).

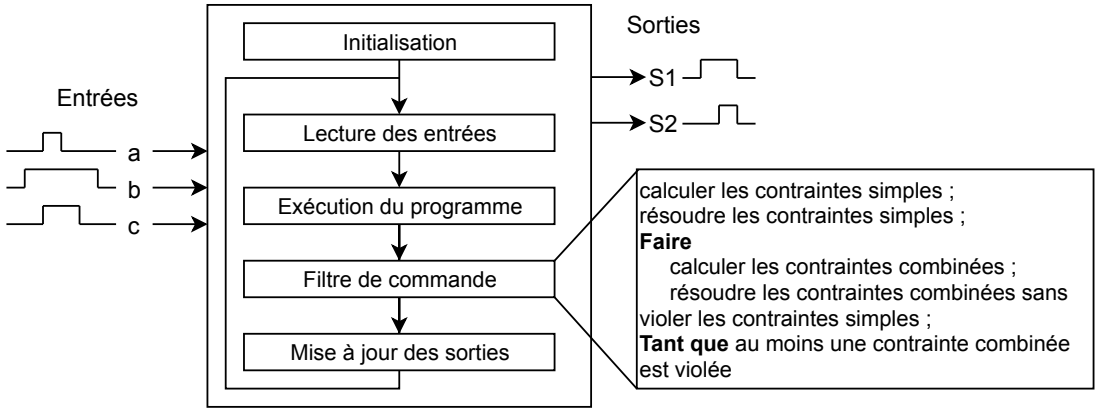


FIGURE 44 – Mise à jour de l'algorithme itératif

IV-2.1 Principe de filtrage par masques logiques

Les masques logiques sont couramment utilisés pour le traitement de vecteurs binaires. De façon générale, un masque permet de modifier un sous-ensemble de variables d'un vecteur binaire en appliquant les opérateurs logiques (ET, OU, NON...) bit à bit. Considérons un bit x quelconque, les règles suivantes sont à appliquer :

ET					OU				
x	.	0	=	0	x	+	0	=	x
x	.	1	=	x	x	+	1	=	1

L'application d'un masque à un mot binaire quelconque, va donc permettre de forcer à 0 certaines variables en utilisant un opérateur ET logique bit à bit, ou de forcer à 1 certaines variables en utilisant un opérateur OU logique bit à bit (Ex. 7).

Exemple 7. *Considérons un mot binaire de 4 bits quelconque* $B = b_1b_2b_3b_4$*, un masque* $M0 = 0101$ *et un masque* $M1 = 0100$*. Le masque* $M0$ *est utilisé avec l'opérateur ET (forçage à 0) et le masque* $M1$ *est utilisé avec l'opérateur OU (forçage à 1). L'application successive de ces masques est représentée ci-dessous :*

B	b_1	b_2	b_3	b_4	
$M0$	0	1	0	1	
	———	———	———	———	ET
$Temp$	0	b_2	0	b_4	
$M1$	0	1	0	0	
	———	———	———	———	OU
O	0	1	0	b_4	

L'exemple 7 montre comment l'application successive d'un masque en ET et d'un masque en OU, permet de « filtrer » un vecteur binaire en forçant certaines variables et en conservant la valeur des autres. La question à présent est donc de savoir comment construire ces masques, afin de résoudre un ensemble de contraintes logiques violées par une affectation des sorties.

IV-2.2 Cas particulier : algorithme de résolution d'un ensemble de contraintes simples

Nous proposons dans un premier temps d'étudier un cas particulier : la résolution d'un ensemble de contraintes simples. La figure 45 présente le fonctionnement du filtre logique, dans un cycle API, lorsque l'ensemble des contraintes est constitué uniquement par des contraintes simples.

Dans le chapitre précédent, des fonctions de forçage simples ont été définies (Déf. 25), celles-ci indiquent si une variable commandable doit être forcée à 0 ($F0s$) ou à 1 ($F1s$) pour résoudre une ou plusieurs contraintes simples. Ces fonctions peuvent donc être utilisées pour calculer la valeur des masques.

Pour rappel, N_o est le nombre de variables commandables. Le masque simple de mise à 0 ($M0s$), permettant la résolution des contraintes simples sur o_k, est défini comme suit :

$$M0s = \overline{F0s^1}\ \overline{F0s^2}\ \ldots\ \overline{F0s^{N_o}}$$

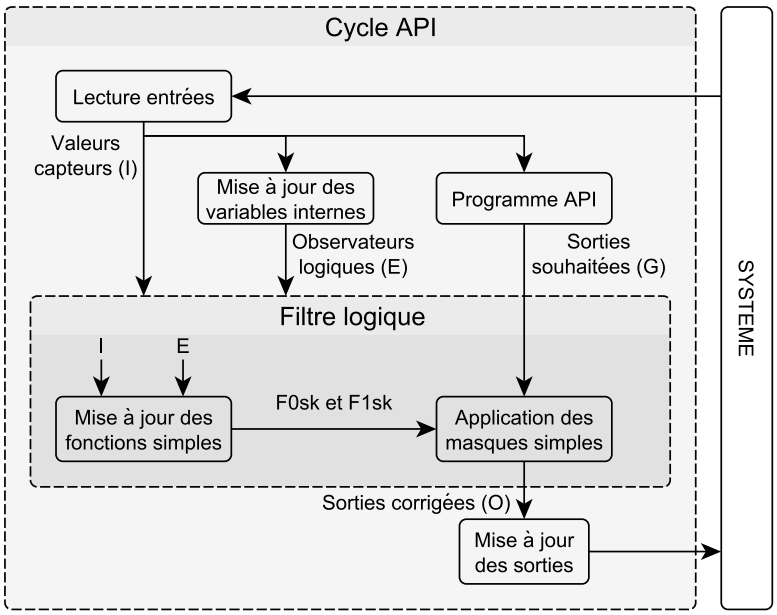


FIGURE 45 – Cycle API avec filtre logique itératif simple

Le masque simple de mise à 1 ($M1s$), permettant la résolution des contraintes simples sur $\overline{o_k}$, est défini comme suit :

$$M1s = F1s^1 \ F1s^2 \ \ldots \ F1s^{N_o}$$

En considérant une affectation quelconque des variables commandables (G), la résolution de l'ensemble des contraintes simples violées par G s'effectue donc en lui appliquant les deux masques simples (Eq. IV.1).

$$\forall k \in \{1, \ldots, dim(G)\}, O_k = \overline{F0s^k}.G_k + F1s^k \tag{IV.1}$$

L'algorithme 1 permet de résoudre un ensemble cohérent de contraintes simples (section III-4.2). Cet algorithme est facilement implémentable en Structured Text (IEC61131-3, 2013) dans un API. Comme rappelé dans la figure 44, cet algorithme est appelé à chaque cycle automate. Il permet de vérifier le vecteur de sorties souhaité, et de le corriger si nécessaire.

A chaque cycle automate, après la mise à jour des variables d'entrées (I) et des variables internes (E, G), le filtre logique est appelé. Premièrement, les fonctions de forçage simples $F0s^k$ et $F1s^k$ sont mises à jour. Puis, à l'aide de ces fonctions, une valeur filtrée de chaque

sortie est calculée pour résoudre les contraintes simples violées (principe des masques).

Il est à préciser que les expressions des fonctions $F0s^k$ et $F1s^k$ ne changent pas d'un cycle à l'autre, seules leurs valeurs sont mises à jours. Ces expressions sont calculées, hors-ligne lors de l'implémentation de l'algorithme, à partir des expressions des contraintes simples (Cf. section III-3.3.1).

Algorithme 1 : Algorithme itératif de résolution d'un ensemble cohérent de contraintes simples

```
Data : No
Input : I, E, G
Output : O
/* Calcul de F0s^k et F1s^k (Déf. 25)                                    */
for k = 1, ..., No do
  | F0s(k) := ... ;
  | F1s(k) := ... ;
end
/* Résolution des contraintes simples                                    */
for k = 1, ..., No do
  | O(k) := NOT F0s(k) AND G(k) OR F1s(k) ;
end
```

La section suivante (section IV-2.3), présente l'algorithme itératif de résolution dans le cas général.

IV-2.3 Cas général : algorithme de résolution d'un ensemble de contraintes

Considérons à présent un ensemble quelconque de contraintes logiques, avec des priorités fonctionnelles associées à chaque contrainte combinée. Considérons également que cet ensemble est cohérent (Déf. 31). Le but est de proposer un algorithme, implémentable dans un API, permettant de résoudre ces contraintes en respectant les priorités. Des masques combinés sont calculés et utilisés pour résoudre les contraintes combinées.

Les masques combinés (M0c et M1c), sont calculés selon le même principe que les masques simples, mais en utilisant les fonctions de forçage combinées :

$$M0c = \overline{F0c^1}\ \overline{F0c^2}\ \ldots\ \overline{F0c^{N_o}}$$

$$M1c = F1c^1\ F1c^2\ \ldots\ F1c^{N_o}$$

La résolution des contraintes combinées, par l'application des masques combinés, ne doit pas remettre en question les modifications effectuées lors de la résolution des contraintes simples (Éq. IV.1). Par conséquent, les masques simples sont utilisés en même temps que les masques combinés, afin de garantir que le vecteur temporaire calculé ne viole pas de contraintes simples (Éq. IV.2).

$$\forall k \in \{1, \ldots, dim(Temp)\}, Temp_k = \overline{F0s^k + F0c^k}.Temp_k + (F1s^k + F1c^k) \qquad \text{(IV.2)}$$

Le principe de résolution est présenté dans la figure 46, celui-ci est une extension du principe de résolution pour les contraintes simples (Fig. 45)

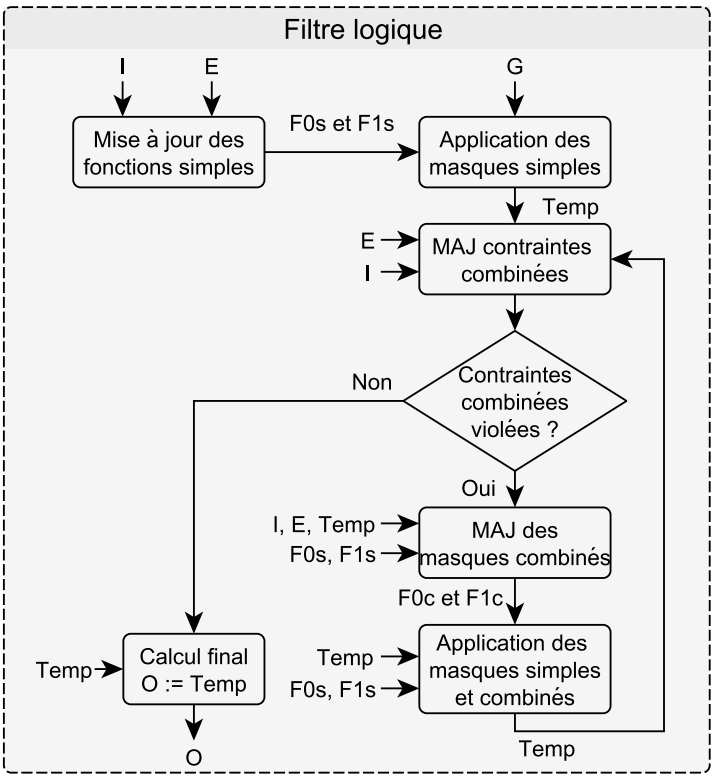


FIGURE 46 – Filtre logique itératif complet

L'application des masques simples, permet l'obtention d'un vecteur de sorties temporaire ($Temp$) satisfaisant les contraintes simples. Ce vecteur temporaire est utilisé pour mettre à jour la valeur des contraintes combinées. Si au moins une contrainte combinée est violée, alors les masques combinés et simples sont utilisés pour calculer un nouveau vecteur $Temp$. Néanmoins, ce vecteur $Temp$ peut violer de nouvelles contraintes combinées. En effet, les

masques combinés sont calculés à partir des fonctions de forçage combinées ($F0c^k$ et $F1c^k$, Déf. 28), et ces fonctions utilisent les valeurs temporaires des sorties ($Temp$). Par conséquent, le vecteur $Temp$ doit être à nouveau testé après avoir été calculé. Dès que le vecteur $Temp$ résout l'ensemble des contraintes, l'algorithme se termine en affectant les valeurs temporaires aux sorties finales.

L'algorithme 2 est la **version complète** de l'algorithme itératif amélioré durant cette thèse.

Algorithme 2 : Algorithme itératif complet

```
Data : No
Input : I, E, G
Output : O
/* Calcul de F0s^k et F1s^k                                          */
for k = 1, ..., No do
    F0s(k) := ... ;
    F1s(k) := ... ;
end
/* Résolution des contraintes simples                                */
for k = 1, ..., No do
    Temp(k) := NOT F0s(k) AND G(k) OR F1s(k) ;
    F0c[k] := false ;
    F1c[k] := false ;
end
Flag := true ;
while Flag do
    Cc := updateCc(Temp, I, E) ;
    Flag := estViolée(Cc) ;
    if Flag then
        for k = 1, ..., No do
            F0c(k) := ... ;
            F1c(k) := ... ;
        end
        /* Résolution des contraintes combinées                      */
        for k = 1, ..., No do
            Temp(k) := NOT(F0s(k) OR F0c(k)) AND Temp(k) OR (F1s(k) OR F1c(k)) ;
        end
    end
end
/* Mise à jour des sorties                                           */
for k = 1, ..., No do
    O(k) := Temp(k) ;
end
```

L'implémentation de cet algorithme en ST (langage API) a été automatisé dans le logiciel SEDMA (Annexe II-3.3).

Étant donné que l'ensemble de contraintes est supposé cohérent (car vérifié auparavant), cet algorithme possède obligatoirement une solution quelles que soient les valeurs des

entrées. Sous les hypothèses de cohérence et d'absence de défaillance des capteurs de la partie opérative, la présence d'une boucle *while* n'est donc pas un problème. En effet, une solution existe donc le « chien de garde » de l'API ne sera pas activé. De plus, étant donné que les expressions des fonctions et des contraintes sont calculées hors-ligne, l'exécution en-ligne de cet algorithme consomme très peu de temps de calcul. Néanmoins, l'utilisation de cet algorithme présente deux inconvénients.

Premièrement, un choix arbitraire de résolution doit être effectué pour résoudre une contrainte combinée : les priorités fonctionnelles (section III-3.3.2). Ce choix restreint par conséquent l'espace de solution atteignable.

Deuxièmement, l'utilisation des priorités fonctionnelles rend complexe la vérification de la cohérence (section III-4). En effet, les priorités fonctionnelles impliquent la construction du graphe d'atteignabilité et son analyse pour statuer sur la cohérence. Ces temps de calculs peuvent être longs. A contrario, la vérification de la cohérence algébrique (sans prendre en compte ces priorités) est assez simple (Déf. 30).

Dans la suite de ce chapitre (section IV-3), nous proposons deux algorithmes, ne nécessitant pas de priorités fonctionnelles pour la résolution d'un ensemble de contraintes logiques.

IV-3 Commande par filtre logique à base de techniques SAT

Cette section a pour but de montrer comment nos travaux peuvent être rapprochés des méthodes de programmation par contraintes de la communauté informatique. Dans un premier temps, la section IV-3.1 montre, à l'aide d'une preuve de concept, comment il est possible d'utiliser un solveur de contraintes pour la commande d'un SED contrôlé par un API. Enfin, dans les sections IV-3.2 et IV-3.3, nous proposons deux algorithmes à base de techniques SAT, implémentables dans un API.

IV-3.1 Filtre logique par solveur SAT : preuve de concept

Avant de chercher à développer un solveur SAT implémentable dans un API, nous avons proposé une preuve de concept, pour la commande par filtre logique à l'aide d'un solveur SAT existant (Pichard *et al.*, 2016). Le principe est d'utiliser un simulateur de partie opérative, connecté à un programme permettant à la fois d'intégrer un programme API, ainsi qu'un solveur SAT. La section IV-3.1.1 présente comment un ensemble de contraintes logiques peut être défini comme un problème SAT-CNF. Puis, la section IV-3.1.2 présente la preuve de concept.

IV-3.1.1 Écriture des contraintes logiques en problème SAT-CNF

Les contraintes logiques définies dans cette thèse sont au format DNF. En effet, les contraintes sont des conjonctions de littéraux (Déf. 17), et ces contraintes sont liées par une disjonction (Éq. III.7).

Un ensemble de contraintes logiques est défini comme **satisfait**, lorsque l'ensemble des contraintes est satisfait (Déf. 24). De plus, une contrainte est satisfaite lorsqu'elle est évaluée à *faux*. Par conséquent, en reprenant la notation utilisée dans la définition 11 (problème SAT satisfaisable), il est possible de montrer qu'un ensemble de contraintes logiques est équivalent à un problème SAT-CNF :

$$\begin{aligned}
\text{Filtre logique satisfait} &\Leftrightarrow \overline{\sum_{i=1}^{dim(C)} \prod_{j} \ell_j = 0} \\
&\Leftrightarrow \sum_{i=1}^{dim(C)} \prod_{j} \ell_j = 1 \\
&\Leftrightarrow \prod_{i=1}^{dim(C)} \sum_{j} \overline{\ell_j} = 1 \\
&\Leftrightarrow \text{SAT-CNF satisfaisable}
\end{aligned}$$

Avec cette transformation, il est donc possible d'utiliser un solveur SAT-CNF pour implémenter le filtre logique. Néanmoins, le filtre logique, et donc le solveur SAT-CNF, a pour objectif d'être utilisé en « temps réel » dans un API afin de commander un système de production. Il est donc essentiel de garantir qu'il existera toujours une solution au problème. Dans le cas contraire, l'API se retrouvera en défaut à cause du déclenchement d'un *chien de garde*. Par conséquent, les valeurs finales envoyées au système ne peuvent pas être

garanties correctes. Néanmoins, si l'ensemble de contraintes est cohérent algébriquement, l'existence d'une solution est garantie.

Dans le cas de l'utilisation d'un solveur SAT pour la commande par filtre logique, les priorités n'ont pas à être utilisées. En effet, celles-ci permettent de forcer la manière de résoudre une contrainte combinée (résolution « statique »). Or, dans le cas d'un solveur SAT, ce sont les techniques de propagations et les heuristiques utilisées, qui permettront de choisir une solution de façon « dynamique » pour les contraintes combinées. Par conséquent, il est possible de garantir qu'il existera toujours une solution (affectation des sorties), quelles que soient les affectations en entrées, en vérifiant la propriété de cohérence algébrique (Déf. 30).

IV-3.1.2 Utilisation d'un solveur SAT en python pour la commande par filtre logique

Le laboratoire CReSTIC collabore depuis 2008, dans le cadre d'un partenariat de recherche et développement, avec la société Real Games [1]. Cette société propose des logiciels de simulations 3D de parties opératives (ITS PLC, FACTORY I/O, HOME I/O). Ces logiciels ont la particularité de pouvoir être connectés simplement à un environnement extérieur, pour le contrôle des parties opératives (programme C#, Python, automate programmable...). La preuve de concept proposée utilise le logiciel ITS PLC connecté à un programme Python.

Différents solveurs CSP et SAT sont disponibles en Python : logilab-constraint [2], pycosat [3]... Logilab-constraint est développé entièrement en Python, contrairement à d'autres (comme pycosat) qui sont des interfaces pour d'autres solveurs développés dans des langages différents. Logilab-constraint permet de définir simplement les variables et les contraintes. De plus, il est possible de spécifier, pour une contrainte donnée, un sous-ensemble de variables modifiables, les autres variables seront considérées comme constantes pour la résolution. Enfin, logilab-constraint est un solveur CSP, mais il est possible de l'utiliser en tant que solveur SAT.

1. `https://realgames.co/`
2. `https://www.logilab.org/project/logilab-constraint`
3. `https://pypi.org/project/pycosat/`

L'utilisation de ces outils logiciels pour réaliser cette preuve de concept est présentée sur la figure 47.

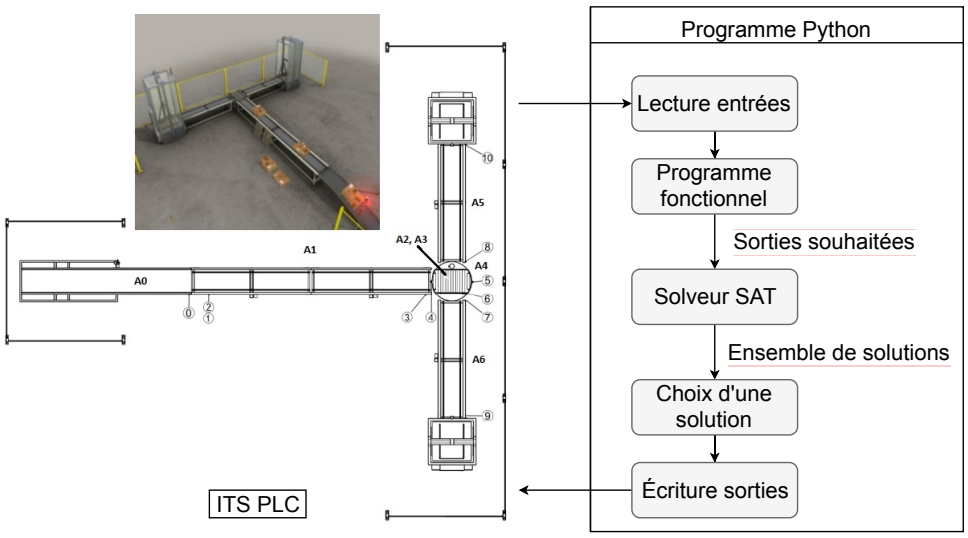


FIGURE 47 – Partie opérative simulée contrôlée par un programme Python

Le système utilisé est un tri de caisses, il est instrumenté de 11 capteurs et 7 actionneurs. Une description complète du système, le programme fonctionnel spécifié en Grafcet, et les contraintes logiques garantissant la sécurité du système sont disponibles dans Pichard *et al.* (2016). La communication entre le programme Python et ITS PLC est cyclique. c'est-à-dire que suite à une lecture des entrées (capteurs), le programme Python s'exécute entièrement, puis les sorties (valeurs actionneurs) sont envoyées au simulateur.

Le principe est donc de calculer, à chaque cycle, un vecteur de sorties souhaitées respectant les spécifications fonctionnelles. Puis, considérant les valeurs mises à jour des entrées et des observateurs logiques, d'utiliser un solveur SAT pour calculer l'ensemble des vecteurs de sorties ne violant aucune contrainte logique. Enfin, un choix parmi l'ensemble des solutions du solveur doit être effectué. Nous proposons d'utiliser la distance de Hamming, entre le vecteur de sorties souhaitées et chacune des solutions du solveur, afin de fournir un indicateur.

La distance de Hamming entre deux mots binaires, est un entier. Cet entier indique le nombre de bit différents entre les deux mots. Par conséquent, plus la distance de Hamming est faible, plus la solution du solveur est proche du vecteur souhaité. Le vecteur final de sorties, envoyé au système, est donc la solution du solveur ayant la distance de Hamming la plus faible avec les sorties souhaitées. En cas d'égalité, la première calculée est utilisée.

Exemple 8. ***Distance de Hamming***

- *Hamming(1011101, 1001001) = 2 ;*
- *Hamming(001, 001) = 0 ;*

Cette heuristique de choix a été utilisée pour sa facilité d'utilisation et d'interprétation. De plus, il peut être considéré que le vecteur souhaité est proche d'une solution. En effet, celui-ci provient d'un programme fonctionnel (intégrant les spécifications fonctionnelles) réalisé par un automaticien.

Le programme Python complet, pour cet exemple, est disponible dans l'annexe IV-1. Voici la structure globale de ce programme :

- initialisation des variables et calcul des fronts associés aux entrées/sorties ;
- définition de la fonction de choix (distance de Hamming) ;
- traduction du Grafcet ;
- calcul des observateurs logiques ;
- calcul des fonctions de forçage simples ;
- calcul du vecteur de sorties souhaitées ;
- ajout des variables et des contraintes au solveur ;
- calcul de l'ensemble des solutions par le solveur ;
- choix de la solution finale ;
- mise à jour des sorties à envoyer au système.

Le temps de cycle maximum, enregistré pour cet exemple, et de 16ms. C'est un temps de cycle cohérent avec les temps de cycles classiques des API, ainsi qu'avec la dynamique du système utilisé. Le fonctionnement observé du système semble très proche de celui observé, avec l'implémentation du filtre logique présentée précédemment (algorithme itératif et priorités). Néanmoins, différents avantages (+) et inconvénients (-) sont à souligner à l'issue de cette preuve de concept :

Avantages :

+ le critère de choix d'une solution n'est pas basé sur des priorités arbitraires entre actionneurs ;

→ l'heuristique de choix peut permettre de prendre en compte des optimisations globales liées à des problématiques de production

\+ étant donné que les priorités ne sont pas nécessaires, le problème de la cohérence des contraintes est restreint à la cohérence algébrique (Déf. 30) ;

→ la phase de conception des contraintes est donc accélérée car la vérification de la cohérence algébrique est moins complexe que la cohérence globale

Inconvénients :

\- le solveur utilisé est en Python et utilise les ressources de calcul d'un ordinateur classique ;

→ un API possède des ressources de calcul inférieur à un ordinateur classique, le temps de cycle risque donc d'être plus long sur un API

→ il n'existe pas, à notre connaissance, de solveur CSP ou SAT implémentable dans un des 5 langages normalisés des API

\- le vecteur de sorties souhaitées est utilisé après avoir calculé l'ensemble des solutions du solveur

→ même si la première solution calculée est la meilleure, l'ensemble des solutions sont calculées

Dans la suite de cette section IV-3, sont présentées deux propositions (sections IV-3.2 et IV-3.3) permettant d'implémenter un solveur SAT dans un API. Les deux propositions sont basées sur une technique de recherche locale, c'est-à-dire qu'ils partent d'une première affectation des variables afin de trouver une solution (Selman *et al.*, 1995). Ces propositions utilisent également la distance de Hamming comme heuristique de choix. La différence fondamentale entre ces deux propositions est l'utilisation du back-tracking pour la seconde (section IV-3.3) et non pour la première (section IV-3.2).

IV-3.2 Algorithme de recherche locale basée sur la distance de Hamming

Cet algorithme de recherche locale, ainsi qu'une implémentation en ST (IEC61131-3, 2013), ont été présentés dans Pichard *et al.* (2018a). L'objectif est de tester, à chaque cycle automate, une affectation des variables de sorties (vecteur G).

Le but de cet algorithme est de trouver un vecteur (*solution*), à partir d'une affectation initiale (FOV) satisfaisant toutes les contraintes. Il est à noter que cet algorithme est appelé à chaque cycle automate. Par conséquent, son exécution doit être rapide (quelques millisecondes). Si au moins une contrainte est violée par G, de nouvelles valeurs doivent être calculées pour les actionneurs (vecteur *solution*). Les valeurs des capteurs, et des observateurs logiques, sont groupées dans le vecteur Y.

L'idée principale est de trouver des valeurs de sorties les « plus proches » du vecteur G (i.e. changeant le minimum de valeurs). Ceci est effectué en utilisant la distance de Hamming. En effet, l'algorithme commence par tester tous les vecteurs ayant une distance de Hamming égale à 1, puis 2, puis 3, etc.

Afin d'améliorer l'efficacité de la recherche, deux vecteurs sont testés simultanéments : *closest* (Hamming distance minimale), *farthest* (Hamming distance maximale). Le vecteur *farthest* est obtenu en complémentant le vecteur *closest*. Si le vecteur *closest* résout le problème, l'algorithme est arrêté et *closest* est utilisé comme solution. Sinon, si *farthest* résout le problème, il est sauvegardé et la recherche recommence avec deux nouveaux vecteurs.

Avec cette approche, si l'espace d'état à explorer comporte N états, seuls $\frac{N}{2}$ boucles seront nécessaires. A la fin de ces boucles, si le vecteur *closest* ne permet pas de résoudre le problème, alors le dernier vecteur *farthest* mémorisé est utilisé comme solution. En effet, ce dernier vecteur *farthest* mémorisé est celui avec la distance de Hamming la plus faible. L'algorithme 3 synthèse cette proposition.

A titre d'exemple, une implémentation de cet algorithme, en langage ST, est disponible en annexe IV-2. L'exemple utilisé est le même tri de caisses que celui utilisé dans la section précédente (section IV-3.1). L'implémentation proposée intègre le programme Grafcet traduit en ST, les contraintes ainsi que les fonctions de forçage. Enfin, certaines fonctions sont définies pour alléger l'écriture, celles-ci sont listées et détaillées à la fin de l'implémentation. Le programme ST détaillé en annexe IV-2, a été implémenté dans un API réel (Schneider Electric M340). La plus grande distance de Hamming enregistrée pour cet exemple, entre le vecteur d'entrée et le vecteur de sortie, a été de 2. Enfin, l'exécution du programme de résolution a toujours été inférieur à 1 ms (Pichard *et al.*, 2018a).

Algorithme 3 : Recherche locale à base de distance de Hamming

```
Input : G, Y
Output : solution
Calculer les valeurs des F0s et F1s ;
solution := initialiser la solution en appliquant F0s et F1s au vecteur G ;
Tester les contraintes combinées (Cc) avec la solution ;
if au moins une Cc est violée then
    index := trouver les index des variables commandables non-forcées ;
    for k = 1 à dim(index)/2 do
        closest := calculer le premier vecteur le plus proche de G ;
        Tester les Cc avec closest ;
        if au moins une Cc est violée then
            farthest := calculer le premier vecteur le plus loin de G en inversant closest ;
            Tester les Cc avec farthest ;
            farthestSolution := stocker farthest si il est solution (Cc résolues) ;
            repeat
                if un nouveau vecteur à tester existe then
                    closest := calculer le vecteur le plus proche suivant ;
                    Tester les Cc avec closest ;
                    if au moins une Cc est violée then
                        farthest := calculer le vecteur le plus loin de G en inversant closest ;
                        Tester les Cc avec farthest ;
                        farthestSolution := stocker farthest si il est solution (Cc résolues) ;
                    end
                end
                if closest résout les Cc then
                    solution := closest ;
                    exit ;
                end
            until le problème est résolu ou tous les vecteurs possibles ont été testés;
        end
    end
    if closest ne résout pas les Cc then
        if farthest résout les Cc then
            solution := farthest ;
        else
            solution := inverser dans G les valeurs des variables commandables non-forcées ;
        end
    end
end
return solution ;
```

Pour conclure, cet algorithme permet de trouver une affectation des variables de sorties, la plus proche au sens de la distance de Hamming, d'une affectation initiale des variables. Étant donné que seules les contraintes sont utilisées (sans les priorités), il suffit de vérifier la cohérence algébrique (Déf. 30), afin de garantir qu'il existera toujours une solution.

Néanmoins, cet algorithme n'utilise aucune technique « classique » utilisée par les solveurs SAT : propagation de contraintes, back-tracking, apprentissage de clauses... Par conséquent, dans le pire des cas l'ensemble de la table de vérité doit être parcourue, la complexité est alors exponentielle par rapport au nombre de variables. Dans la section suivante (section IV-3.3), nous proposons un algorithme, implémentable dans un API, utilisant de la propagation de contraintes et du back-tracking afin d'optimiser la recherche d'une solution.

IV-3.3 Solveur SAT pour automate programmable industriel

L'objectif reste le même que précédemment : l'algorithme doit être capable de résoudre, en-ligne dans un API, un ensemble de contraintes logiques violées par une affectation initiale des sorties. Néanmoins, dans le but de gagner en efficacité, les techniques de back-tracking et de propagations de contraintes sont intégrées.

L'intégration du solveur SAT dans le cycle API est présenté dans la figure 48. Dans un premier temps, les masques simples sont utilisés pour résoudre les contraintes simples. Ensuite, le vecteur résultant ($Temp$) est donné comme entrée du solveur si au moins une contrainte combinée reste violée.

L'algorithme 4 décrit globalement le bloc « Solveur SAT » de la figure 48.

L'implémentation complète de cet algorithme, en ST dans un API, est fournie en annexe IV-3. Comme précédemment dans cette section, l'exemple utilisé pour cette implémentation est celui du tri de caisses (Fig. 47).

La première étape de l'algorithme est son initialisation (annexe IV-3.1). Le but est d'initialiser, à chaque cycle automate, les variables et tableaux afin d'effectuer proprement la recherche d'une solution. En fonction du problème à résoudre, deux paramètres sont à modifier : $MAX\_NUM\_VARIABLE$ permet d'indiquer le nombre maximum de variables

Algorithme 4 : Solveur SAT pour API avec recherche locale et distance de Hamming

```
Input : Temp, Y
Output : solution
Initialisation de la structure de donnée ;
Ajouts des contraintes violées au problème ;
Initialisation de la recherche ;
while Il existe au moins une solution ET la distance est différente de 1 do
    /* Affectation d'une variable à partir du vecteur Temp (recherche locale)   */
    foreach variable do
        if la variable est libre then
            Affecter la variable picked avec la valeur correspondante de Temp ;
        end
    end
    /* Hamming, mémorisation, redémarrage de la recherche                       */
    if une solution a été trouvée (pas de nouvelles variables affectées) then
        mem := mémorisationSolution() ;
        Ajout de la solution, sous forme complémentée, à l'ensemble des contraintes ;
        dist := Hamming(Temp, mem) ;
        if dist < distPrecedent then
            distPrecedent := dist ;
        end
        if distPrecedent ≠ 1 then
            backtracking() ;
        end
    end
    if dist ≠ 1 then
        Mise à jour de la structure de donnée ;
        Ajout de la variable picked à la pile ;
        /* Propagation de contraintes                                           */
        while une variable est à affecter ET un conflit n'est pas détecté do
            var_to_set := prendre une variable dans la pile ;
            Satisfaire les contraintes contenant var_to_set ;
            Réduire les contraintes contenant ¬var_to_set ;
            /* Vérifier les contraintes                                         */
            foreach contraintes do
                /* Conflit et back-tracking                                     */
                if la contrainte est totalement affectée et non résolue then
                    backtracking() ;
                end
                /* Gestion des contraintes unitaires                            */
                if la contrainte est unitaire then
                    Ajout de la dernière variable de la contrainte à la pile ;
                end
            end
        end
    end
end
solution := mem ;
```

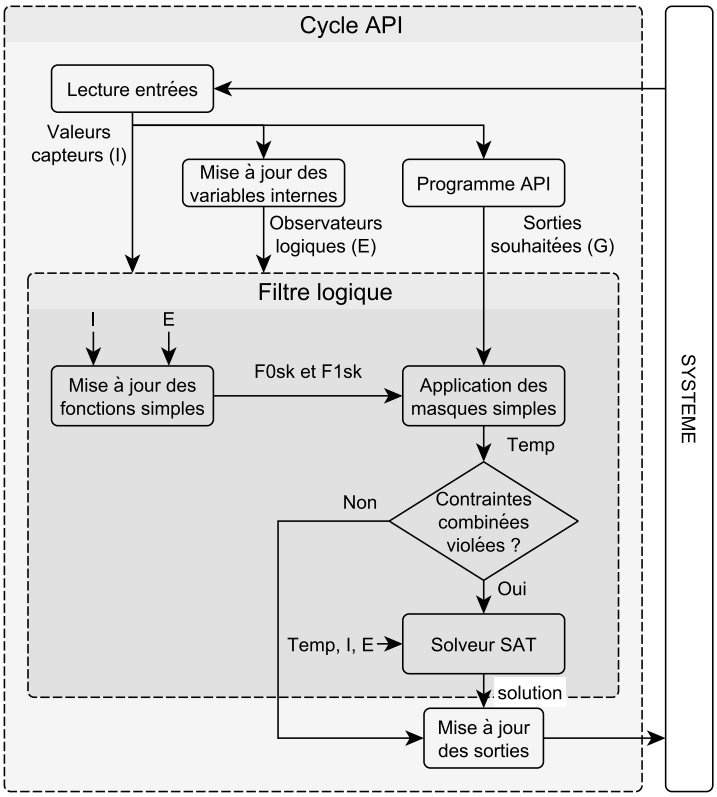


FIGURE 48 – Cycle API avec solveur SAT

dans une contrainte, MAX_NUM_CLAUSE permet d'indiquer le nombre maximum de contraintes utilisables. Ces paramètres servent à définir la taille de certains tableaux, ainsi que les bornes de parcours de certaines boucles.

L'étape suivante est l'ajout des contraintes à résoudre au problème (annexe IV-3.2). Afin d'optimiser la recherche de solutions, seules les contraintes dont la partie non-commandable est évaluée à *true* sont ajoutées au problème. De plus, seules les parties commandables de ces contraintes sont ajoutées au problème. Dans ces conditions, seules les variables commandables seront utilisées pour la résolution du problème, ce qui diminue fortement le nombres de variables à gérer. En effet, dans un système de production automatisé, le nombre de capteurs et de variables internes est souvent bien supérieur au nombre d'actionneurs.

La structure de données utilisée pour implémenter cet algorithme en ST, se base sur 3 tableaux principaux : *formula*, *var_in_clause* et *status*.

— status[MAX_NUM_VARIABLE] = {UNDEF, TRUE, FALSE} : permet de stocker l'état de chaque variable (UNDEF signifie que la variable n'a pas été affectée).

— formula[MAX_NUM_CLAUSE][MAX_CLAUSE_SZ] : permet de stocker pour chaque contrainte, quelles sont ses variables. De plus, MAX_CLAUSE_SZ = MAX_NUM_VARIABLE+3. Le « +3 » permet de stocker différentes informations à propos de la contrainte (est-elle satisfaite, est-elle assignée, quelle est la taille initiale de la contrainte).
— var_in_clause[MAX_NUM_VARIABLE*2+1][MAX_NUM_CLAUSE] : matrice permettant de savoir dans quelle contrainte est chaque variable.

Le codage utilisé pour les littéraux est le suivant : à chaque littéral l est associé un entier (différent de 0). Le signe de cet entier indique si le littéral est sous forme normale ($+l$) ou sous forme complémentée ($-l$).

Les matrices *var_in_clause* et $formula$ peuvent sembler redondantes. En effet, initialement ces deux matrices contiennent le même type d'informations : quelles variables sont dans quelles contraintes. Néanmoins, les statuts de chaque contrainte dans $formula$ vont évoluer pendant la recherche de solution, alors que *var_in_clause* ne sera pas modifié pendant la recherche. De plus, *var_in_clause* permet de savoir facilement dans quelles contraintes apparaît une variable, alors que $formula$ permet de savoir quelles variables sont dans une contrainte. Pour finir, *var_in_clause* est utilisé afin de trouver efficacement quelles contraintes sont à modifier suite à l'affectation d'une variable, ainsi que pour reconstruire $formula$ et défaire des affectations lors d'un back-tracking.

Afin de pouvoir lire sereinement l'implémentation proposée en annexe IV-3, la structure de *var_in_clause*[*ligne*][*colonne*] mérite d'être expliquée :

— chaque ligne i correspond à une variable propositionnelle v_i, il en existe donc 2 fois plus que de littéraux ;
— la première colonne de chaque ligne indique dans combien de contraintes apparaît v_i ;
— les colonnes suivantes contiennent les indices des contraintes contenant v_i.

L'exemple 9 détaille la définition de cette matrice pour un ensemble de contraintes.

Exemple 9. *Considérons un ensemble de littéraux* $L = \{a, b, c, d\}$*, et un ensemble de contraintes* $C = \{C1, C2, C3\}$ *telles que :*

$C1 : a.\overline{b} = 0 \quad \Leftrightarrow \quad \overline{a} + b = 1;$

$C2 : a.b.\overline{d} = 0 \quad \Leftrightarrow \quad \overline{a} + \overline{b} + d = 1;$

$C3 : a = 0 \quad \Leftrightarrow \quad \overline{a} = 1.$

Alors, avec M = *MAX_NUM_VARIABLE, la matrice var_in_clause peut se représenter sous cette forme :*

$\overline{d}$ *(M-4)*	*1*	*2*
$\overline{c}$ *(M-3)*	*0*	
$\overline{b}$ *(M-2)*	*1*	*1*
$\overline{a}$ *(M-1)*	*0*	
M		
a (M+1)	*3*	*1 2 3*
b (M+2)	*1*	*2*
c (M+3)	*0*	
d (M+4)	*0*	

Il est donc simple de savoir que a apparaît dans les 3 contraintes d'indice 1, 2 et 3, alors que $\overline{a}$ *n'apparaît dans aucune contrainte.*

Pour finir, le cœur de l'algorithme de recherche (Alg. 4) est fourni en annexe IV-3.3. Par soucis de lisibilité, la fonction de back-tracking est contenue dans une fonction détaillée dans l'annexe IV-3.4.

La section suivante a pour objectif de fournir des éléments de comparaison pour les différents algorithmes proposés.

IV-4 Synthèse et comparaison des algorithmes proposés

Dans ce chapitre IV, différents algorithmes ont été proposés afin d'implémenter un filtre logique dans un API. Dans cette section IV-4, le terme algorithme *itératif* fera référence à la première proposition (Alg. 2), algorithme de *Hamming* fera référence à la seconde

proposition (Alg. 3), et enfin, *solveur SAT* fera référence à la troisième proposition (Alg. 4). L'algorithme 1, ne référant qu'aux contraintes simples, ne peut pas faire l'objet de cette comparaison (il est inclus dans l'algorithme *itératif*).

Avant de les comparer, un résumé des caractéristiques principales de ces algorithmes est présenté ci-dessous :

- Alg. itératif : algorithme « historique » du filtre logique, peu complexe, hypothèse de cohérence des contraintes avec priorités ;
- Alg. Hamming : hypothèse de cohérence algébrique uniquement, technique de recherche locale basée sur la distance de Hamming ;
- Solveur SAT : hypothèse de cohérence algébrique, techniques SAT (propagation de contraintes, back-tracking), recherche locale, heuristique de choix (Hamming).

Dans le but de pouvoir comparer ces différents algorithmes, plusieurs critères de comparaison sont proposés ci-dessous. Par la suite, la comparaison est effectuée dans le tableau 16.

Priorités L'utilisation des priorités permet de forcer la résolution d'une contrainte combinée. L'espace des solutions atteignables est donc plus petit que l'espace des solutions. De ce fait, certaines solutions ne seront jamais utilisées. Dans certains cas, cela peut impliquer un blocage du système. Ces priorités sont à choisir lors de la phase de conception des contraintes, ce qui allonge le temps de conception.

L'algorithme *itératif* a besoin de ces priorités pour trouver une solution, ces fonctions de résolution sont entièrement basées sur ces priorités. Les autres algorithmes n'ont pas besoins de ces priorités pour trouver des solutions.

Cohérence La vérification de la cohérence de l'ensemble des contraintes, permet de garantir qu'il existera toujours une solution quelles que soient les entrées. Néanmoins, cette vérification demande du temps. Lorsque les priorités ne sont pas utilisées, la cohérence algébrique suffit à garantir l'existence de solutions. Enfin, la vérification de la cohérence algébrique est bien moins complexe, en temps de calcul, que la vérification de la cohérence « globale ».

L'algorithme *itératif* nécessite une vérification complète de la cohérence (avec les prio-

rités), alors que les autres algorithmes ne nécessitent que la vérification de la cohérence algébrique. Le temps de conception est donc plus rapide avec les algorithmes *Hamming* et *Solveur SAT*

Implémentation La difficulté d'implémentation de ces algorithmes impacte directement leurs utilisations. L'évaluation de ce critère est lié aux nombres de lignes de codes, aux nombres de boucles et d'imbrications de ces boucles. La structure de données utilisée est également un indicateur important. En effet, un tableau à plusieurs dimensions est plus difficile à appréhender et gérer que de simples variables.

L'algorithme *itératif* est très simple à implémenter, en effet il suffit d'une boucle principale pour résoudre le problème. De plus, la structure de données utilisée ne comporte que quelques tableaux et variables.
L'algorithme *Hamming* est quand à lui plus compliqué à implémenter. En effet, celui-ci nécessite de nombreuses boucles et tests. De plus, plusieurs variables supplémentaires sont nécessaires par rapport à l'algorithme *itératif*.
Enfin, le *solveur SAT* est le plus difficile à implémenter à cause des techniques de propagation de contraintes et de back-tracking. La structure de données comporte des matrices et de très nombreuses variables.

Maintenabilité La maintenabilité d'un programme API est fortement liée à la lisibilité du code. Dans un contexte de production industrielle, il est important de pouvoir suivre l'évolution du programme.

L'algorithme *itératif* permet de suivre en « temps réel », la résolution des contraintes. En effet, il est assez simple de ressortir les informations liées aux contraintes violées, aux variables forcées, et aux priorités actives. Dans les deux autres algorithmes, cela est bien plus compliqué. En général, seuls les vecteurs d'entrées et de sorties pourront être observés. La lisibilité et le suivi du déroulement de la résolution sont donc compliqués.

Espace mémoire Un API comporte en général quelques Mo de mémoire de travail et quelques Go de mémoire de stockage. Il est donc important que les algorithmes n'engendrent pas la création de trop de variables et tableaux.

L'espace mémoire utilisé par les algorithmes est fortement lié à la structure de données discutée précédemment (Cf. **Implémentation**). Par conséquent, l'algorithme *itératif* est le plus léger et le *solveur SAT* est le plus lourd.

Adaptabilité Les systèmes de production sont tous différents. Il peut alors être important de pouvoir adapter les programmes, afin de correspondre au mieux aux besoins. Un algorithme faiblement adaptable peut restreindre son application à tous types de systèmes. A contrario, trop de paramètres à régler peut rendre complexe son utilisation.

Les algorithmes *itératif* et *Hamming* ne sont pas ajustables. C'est-à-dire qu'ils dépendent exclusivement des contraintes (et des priorités dans le cas de l'*itératif*) afin de trouver une solution. Le *solveur SAT* peut être considéré comme plus adaptable. En effet, il est possible de modifier le critère d'arrêt (dans notre cas une minimisation de la distance) ainsi que l'heuristique de choix entre 2 solutions (dans notre cas la distance de Hamming). La modification de ces deux points permet le changement de dynamique de l'algorithme, sans avoir à changer sa structure et son fonctionnement.

Génération automatique Dans le but de limiter au maximum les erreurs de programmations, il est préférable d'automatiser au maximum l'écriture des programmes (implémentation). La génération automatique d'un programme automate, à partir de données d'entrées, est donc un point important afin de limiter les temps de conception et le risque d'erreurs. Néanmoins, dans un contexte de production et de maintenance d'un système industriel, il est préférable de pouvoir modifier le programme API sans avoir à passer par un logiciel de génération automatique. En effet, si la modification ou l'ajout d'une contrainte implique de devoir re-générer entièrement l'implémentation de l'algorithme, cela est une perte de temps.

Dans le cas du filtre *itératif*, les contraintes simples n'apparaissent pas explicitement dans l'implémentation, celles-ci sont incluses dans les fonctions de forçage simples. De même, les priorités des contraintes combinées sont incluses dans les fonctions de forçage combinées. La modification d'une contrainte simple, d'une contrainte combinée ou bien d'une priorité peut engendrer de nombreux changements dans l'implémentation. Il est donc, en général, nécessaire d'utiliser un outil de génération automatique pour le filtre *itératif*. Cet outil

doit prendre, en entrée, un ensemble de variables, de contraintes et de priorités, et fournir automatiquement en sortie un code ST implémentable dans un API.

Pour l'algorithme de *Hamming* et le *solveur SAT*, les contraintes apparaissent explicitement au niveau de l'initialisation des algorithmes : fonctions init_CSC() et calcul_CSC() pour *Hamming* (section IV-2), section IV-3.2 pour le *solveur SAT*. De plus, les priorités ne sont pas utilisées. Par conséquent, la modification ou l'ajout d'une contrainte peuvent être effectués directement dans le programme API sans passer par une étape de génération automatique. L'entièreté de l'implémentation n'est pas à modifier.

Complexité Les complexités théoriques des algorithmes n'ont pas été étudiées dans le cadre de cette thèse. Néanmoins, il est possible d'obtenir des indicateurs de complexité à l'aide de l'outil PLC Checker [4] de la société Itris Automation. Cet outil permet l'analyse d'un programme automate. Différents critères d'analyses sont utilisés, ils sont en général basés sur le standard PLC OPEN [5]. Dans le cadre de cette comparaison, deux métriques différentes sont utilisées : la complexité cyclomatique et la complexité essentielle (McCabe, 1976). La complexité cyclomatique indique le nombre de chemins indépendants possibles avec l'algorithme. Pour un API, il est d'usage d'avoir cette complexité inférieure à 15 (d'après PLC Checker). La complexité essentielle mesure le nombre de points de sortie. Il est d'usage d'avoir une complexité essentielle de 1 pour faciliter la maintenance. Ces métriques sont calculées directement par PLC Checker.

L'algorithme *itératif* est le moins complexe à tout niveau. Les métriques indiquent que l'algorithme *Hamming* est plus complexe que le *solveur SAT*. Nos expérimentations confirment ce résultat lorsqu'une solution unique existe, et que celle-ci correspond à la plus petite des moins bonnes solutions (Cf. vecteur *farthest* de l'algorithme *Hamming*). Néanmoins, lors de l'utilisation de ces algorithmes sur différents exemples concrets (comme le tri de caisses), l'algorithme de *Hamming* est meilleur (en temps de calcul) que le *solveur SAT*.

Ces métriques fournissent donc des premières indications, néanmoins il semble nécessaire de continuer à approfondir cette étude.

4. http://www.itris-automation.fr/plc-checker/
5. http://www.plcopen.org/

TABLE 16 – Comparaison des algorithmes proposés

	Alg. itératif	Alg. Hamming	Solveur SAT
Priorités	oui	non	non
Cohérence	globale	algébrique	algébrique
Implémentation	très simple	difficile	très difficile
Maintenabilité	simple	très difficile	très difficile
Espace mémoire	faible	moyen	très grand
Adaptabilité	aucune	aucune	critère d'arrêt heuristique de choix
Génération automatique	nécessaire	possible	possible
Complexité cyclo. Complexité essen.	8 1	39 4	29 2

IV-5 Conclusion

Ce chapitre a présenté différentes contributions relatives à l'implémentation du filtre logique dans un API.

Tout d'abord, l'algorithme *itératif* a été amélioré en se basant sur les contributions du chapitre III (formalismes, priorités et cohérence). Cet algorithme itératif permet une implémentation simple du filtre logique, mais nécessite l'utilisation des priorités rendant l'analyse de la cohérence des contraintes assez complexe.

Par la suite, deux algorithmes de recherche locale d'une solution, basés sur des techniques de solveur SAT, ont été proposés. Ces algorithmes sont plus complexes à implémenter, mais permettent de ne pas utiliser les priorités pour imposer un choix de résolution des contraintes combinées. Dès lors, la vérification formelle de la cohérence est simplifiée et par conséquent le temps de conception est réduit.

Ces travaux montrent que l'utilisation de solveur SAT pour le contrôle d'un API est possible. Néanmoins, de nombreuses voies d'améliorations sont envisageables (efficacité, structures de données, lisibilité...). Le développement d'un solveur SAT dédié aux API est donc un problème ouvert pour les communautés informatiques et automatiques.

Dans le chapitre suivant, l'approche de commande par filtre logique est appliqué à un système réel.

CHAPITRE V

Application à un système réel : CellFlex

La cellule flexible (Fig. 49) est un élément principal de la plate-forme de formation et de recherche CELLFLEX4.0 de l'Université de Reims Champagne-Ardenne. Cette plate-forme regroupe un ensemble d'outils pour la simulation de systèmes manufacturiers (ITS PLC, Factory I/O) et de maison virtuelle (Home I/O), un atelier flexible (CellFlex) ainsi qu'une Plate-Forme Multi-Énergies Renouvelables (PFMER). La plate-forme CELLFLEX4.0 est issu de l'axe ICOS du CPER 2007-2013 et s'inscrit dans le cadre du projet FFCA (Factory of Future Champagne-Ardenne) du CPER 2017-2022.

FIGURE 49 – La CellFlex

Dans le cadre de cette thèse, l'atelier CellFlex a été utilisé pour les tests et la validation des propositions. Différentes utilisations possibles du filtre logique ont été présentées

précédemment (section III-2.4). Pour cette application, l'approche par **filtre de sécurité** a été retenue. Pour rappel, un filtre de sécurité a pour objectif de ne traiter que la partie sécurité/sûreté du cahier des charges, la partie fonctionnelle du cahier des charges étant supposée inconnue lors de la définition des contraintes. Avec cette approche, les contraintes logiques sont appelées **Contraintes de Sécurité** (CS).

L'utilisation d'un filtre de sécurité sur un système automatisé de formation, comme CellFlex, permet de limiter le risque de casse et d'augmenter l'autonomie des apprenants. En effet, le filtre de sécurité annulera toutes les actions dangereuses, par conséquent l'apprenant peut tester son programme sans avoir besoin de la vérification de l'enseignant. De plus, il est possible de remonter automatiquement à l'apprenant la ou les contraintes violées par son programme. Celui-ci a donc une information concrète des exigences de sécurité non respectées par son programme.

Ce chapitre V présente dans un premier temps le système dans son ensemble (section V-1). Puis dans la section V-2, pour chaque station utilisée, une description du système, le cahier des charges et les contraintes de sécurités sont détaillés. De plus, le filtre de sécurité est utilisé de manières différentes sur ces différentes stations permettant ainsi de montrer les différents avantages de l'approche.

V-1 Description du système

La cellule flexible, appelée CellFlex, est un groupe de huit stations fonctionnant ensemble autour d'un convoyeur central. Ces stations simulent le fonctionnement d'une ligne de production de mise en bouteille, sous la forme d'une usine miniaturisée reliée sur un réseau composé de standards industriels. Une représentation schématique de la cellule est présentée sur la figure 50. Il existe six Automate Programmable Industriel (API) différents pour le contrôle/commande des différentes stations. Chaque API peut donc contrôler plusieurs stations, les numéros présents sur la figure 50 indiquent quel API contrôle quelle(s) station(s). Les flux entre chaque station sont représentés par des flèches.

Une description plus précise des stations et de leur regroupement sur les automates est proposée ci-dessous.

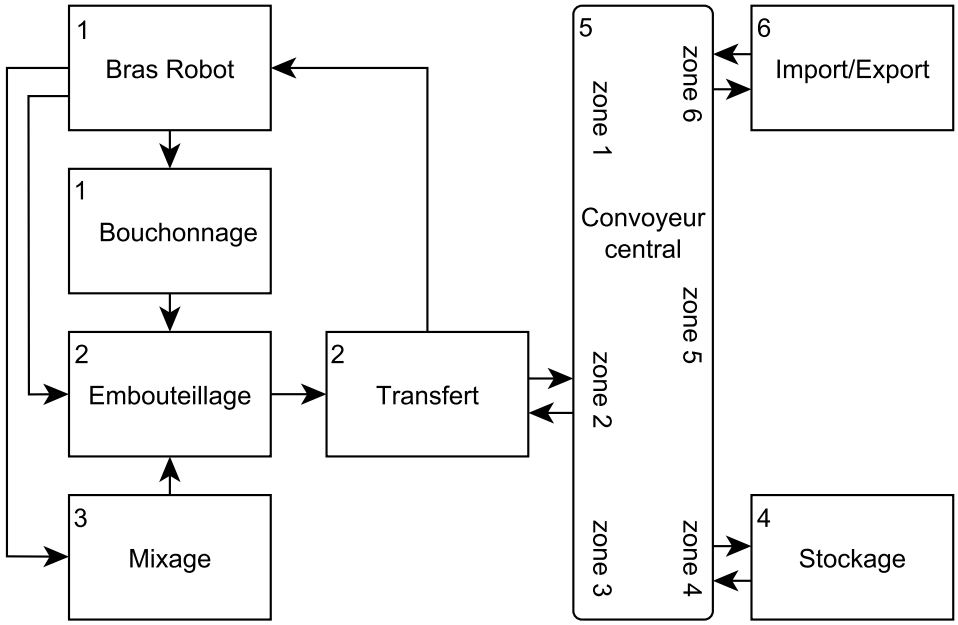


FIGURE 50 – Les stations de la CellFlex

automate 1 : Cet automate regroupe le bras robot et la production de bouchons. Le bras robot permet la récupération de bouteilles pleines. Il transfère les bouchons dans le stock approprié du module de bouchonnage et aspire le liquide pour le rapporter à la station de mixage avant de renvoyer les bouteilles vides à la station d'embouteillage. La station de bouchonnage permet de sélectionner un bouchon en fonction du type souhaité, et de l'acheminer vers l'embouteillage.

automate 2 : Cet automate regroupe la station d'embouteillage et la station de transfert. Les bouteilles vides arrivent sur la table tournante de l'embouteillage où le liquide préparé par le mixeur est injecté. Le bouchonnage amène un bouchon qui est placé sur la bouteille, le bouchon est ensuite vissé puis la bouteille est envoyée vers la station de transfert. La station de transfert permet, à l'aide d'un bras manipulateur 2-axes, de placer 3 par 3 des bouteilles dans un 6-pack (carton pouvant accueillir six bouteilles) en attente sur le convoyeur central.

automate 3 : L'automate 3 comporte uniquement la station de mixage. Celle-ci permet de mélanger jusqu'à trois liquides selon des quantités différentes. Le résultat est ensuite transféré vers la station d'embouteillage.

automate 4 : L'automate 4 contient un stockage vertical. La station de stockage permet de mettre des 6-packs vides ou pleins dans 16 emplacements différents pour une utilisation ultérieure.

automate 5 : Cet automate contrôle le convoyeur central. Celui-ci permet de desservir les stations de transfert, stockage et import/export en 6-packs. Des palettes guidées dans un rail permettent d'accueillir les 6-packs à déplacer. Ce convoyeur central comporte 6 zones distinctes permettant des échanges avec des stations en vis-à-vis.

automate 6 : L'automate 6 comporte la station d'import/export. La station d'import/export permet d'introduire ou d'extraire des 6-packs du système (pleins ou vides) à l'aide d'un bras manipulateur 3 axes.

La CellFlex est équipée entièrement en automates Siemens (un par zone) de gamme S7-300. Certaines zones comportent des blocs d'entrées/sorties déportées et le convoyeur central est associé à un variateur de vitesse micromaster 420. Enfin, chaque station possède une Interface Homme Machine (IHM) permettant un contrôle global ou local de la CellFlex.

Le réseau de la CellFlex est composé de 3 standards principaux : Ethernet, MPI et ModBus. Une description schématique de la structure réseau, entre les automates et les entrées-sorties déportées, est proposée dans la figure 51 (extrait de TIA portal [1]). Les entrées/sorties déportées des différents automates sont reliés aux capteurs et actionneurs des stations par un bus de terrain PROFINET. Pour le convoyeur central, l'automate et la station communique par un bus ASi.

Différents modules RFID permettent de lire et écrire des informations dans des puces RFID présentent dans les bouchons des bouteilles permettant ainsi de la traçabilité. De plus, des cameras sont également présentes à différents endroits de la station permettant la détection de défauts ou la vérification de présence de bouteilles dans un 6-pack. Ces modules, RFID et vision, ne sont pas représentés et ne seront pas utilisés dans ce chapitre.

Il est à noter que les informations du convoyeur central ne sont pas directement accessibles des autres zones. Il est nécessaire de passer par la zone 4 (station stockage) afin d'accéder aux informations du convoyeur.

1. `https://www.industry.siemens.com/topics/global/fr/tia-portal/pages/default.aspx`

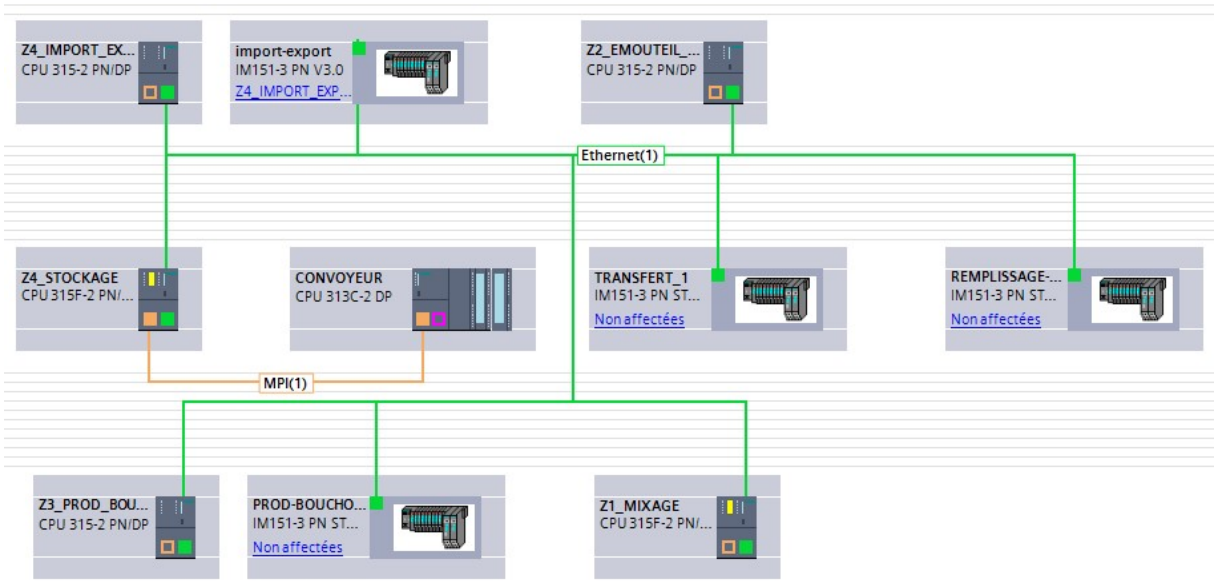


FIGURE 51 – Architecture réseau de la Cellule Flexible

V-2 Application à trois stations

Cette partie applicative de la thèse a permis de tester et valider l'approche par filtre logique.

La suite de ce chapitre présente la mise en place des filtres de sécurité sur 3 stations : le convoyeur central, le transfert et l'import/export. Pour chaque station est détaillé un cahier des charges, puis les exigences de sécurité sont spécifiées à l'aide de contraintes logiques de sécurité. L'analyse formelle de la cohérence des contraintes est effectuée et l'implémentation automatique de l'algorithme du filtre logique est réalisée avec le logiciel SEDMA. De plus, trois applications différentes du filtre de sécurité sont proposées. Ces différentes applications permettent de montrer trois manières de coupler le programme fonctionnel au filtre de sécurité.

V-2.1 Convoyeur central

Le convoyeur central est le système autour duquel s'organisent toutes les stations. Il permet de transporter des 6-packs par le biais de palettes entrainées par des convoyeurs à bandes.

Un ensemble de capteurs est disponible. Pour chaque zone du convoyeur central numérotée de 1 à 6, il existe quatre capteurs différents :

— *palette_Si* : capteur inductif détectant la présence d'une palette dans la zone i ;
— *pack_Si* : capteur infrarouge détectant la présence d'un 6-pack sur la palette ;
— *sortir_Si* : capteur inductif détectant présence d'une palette en sortie de la zone i ;
— *compteur_Si* : capteur inductif permettant l'identification d'une palette par un système de comptage logiciel.

Quatre tapis convoyeurs sont reliés électriquement sur la même sortie automate, et donc contrôlés par une même variable booléenne. Ces tapis permettent l'entraînement des palettes.

De plus, il existe devant chaque station un vérin bloqueur (Ver_Si). Ces vérins bloqueurs permettent d'arrêter les palettes devant une station si nécessaire, mais également de réguler le flux des palettes. Il est à noter que les vérins sont des simples effets dont l'état naturel est « sorti ». L'ordre d'activation les fera donc se rétracter.

Une représentation schématique du convoyeur est décrite sur la figure 52. Le positionnement des différentes stations autour de ce convoyeur a été présenté dans la figure 50.

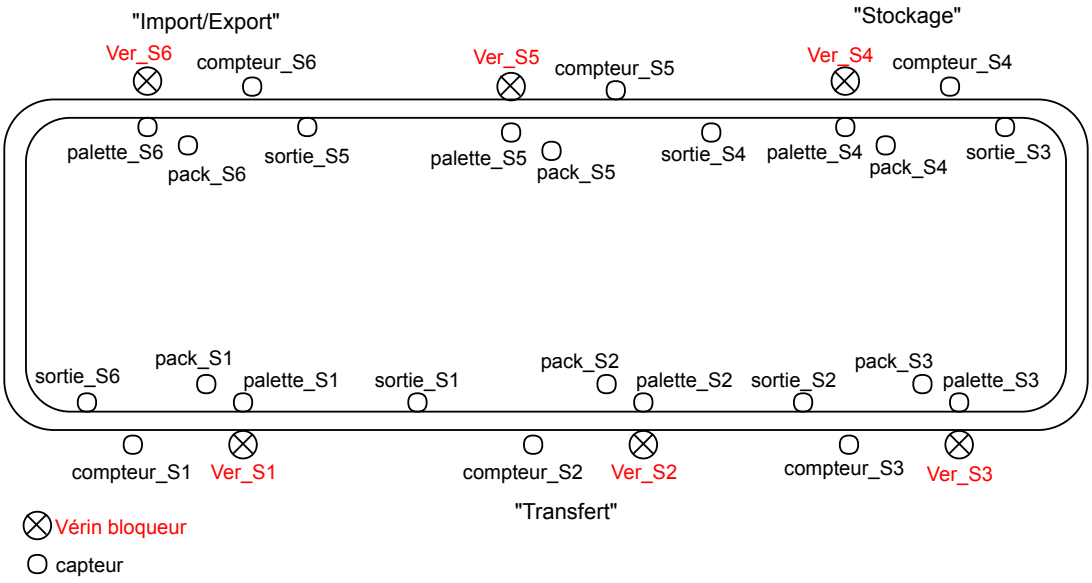


FIGURE 52 – Description du convoyeur central

Une particularité du convoyeur central est que tous les capteurs et actionneurs précédemment décrits sont enregistrés dans un bus AS-i. L'utilisation dans le logiciel TIA Portal

passe par la création de deux tableaux (IN et OUT), regroupés dans un bloc de données (DB), dans lesquels les informations associées aux capteurs et actionneurs sont enregistrées. Un pupitre comportant des boutons et des voyants est également associé à cet automate. Enfin, certaines informations sont issues d'autres stations afin de pouvoir garantir le bon fonctionnement. Le tableau 17 recense les différents capteurs et actionneurs disponibles sur cet automate, ainsi que des variables internes utilisées par les contraintes logiques présentées dans la suite de cette section.

V-2.1.1 Cahier des charges

Le cahier des charges pour la programmation de cette station est assez succinct et peut être décrit à l'aide de ces différentes exigences :

1 les convoyeurs à bandes sont toujours allumés ;
2 une seule palette doit être présente par section (entre les capteurs *palette_Si* et *sortie_Si*) ;
3 la montée d'un vérin bloqueur ne doit pas pouvoir s'effectuer si une palette est présente au-dessus de ce vérin ;
4 les vérins bloqueurs ne doivent pas descendre si le bras de l'import/export ou du transfert sont en mouvement au dessus de la zone.

Il est à noter que la collision entre les palettes n'apparaît pas dans le cahier des charges. En effet, les palettes sont physiquement conçues pour pouvoir rentrer en collision sans risque entre elles ou avec les vérins bloqueurs.

V-2.1.2 Définition des contraintes de sécurité

L'exigence n° 3 est liée à la sécurité du système : « La montée d'un vérin bloqueur ne doit pas pouvoir s'effectuer si une palette est présente au-dessus ».

Cette situation peut se produire si l'ordre d'un vérin est annulé alors qu'une palette se situe encore au-dessus de celui-ci. La spécification logique de cette exigence est alors : « Si une palette est présente alors interdire le front descendant de l'ordre d'activation du vérin ». Étant donné qu'il y a 6 vérins, l'ensemble de contraintes nécessaires pour spécifier

Table 17 – Mnémoniques du convoyeur central

Mnémonique	Type	Adresse	Description
btn_ON_auto	Capteur	%I124.0	Bouton poussoir « ON » Auto
btn_OFF_auto	Capteur	%I124.1	Bouton poussoir « OFF » Auto
btn_reset	Capteur	%I124.2	Bouton poussoir reset
palette_S1	Capteur	INT[2].BIT1	Présence palette zone 1
sortie_S1	Capteur	INT[2].BIT2	Sortie de la zone 1
pack_S1	Capteur	INT[2].BIT3	Présence 6-pack zone 1
compteur_S1	Capteur	INT[2].BIT4	Identification de palette zone 1
palette_S2	Capteur	INT[5].BIT1	Présence palette zone 2
sortie_S2	Capteur	INT[5].BIT2	Sortie de la zone 2
pack_S2	Capteur	INT[5].BIT3	Présence 6-pack zone 2
compteur_S2	Capteur	INT[5].BIT4	Identification de palette zone 2
palette_S3	Capteur	INT[8].BIT1	Présence palette zone 3
sortie_S3	Capteur	INT[8].BIT2	Sortie de la zone 3
pack_S3	Capteur	INT[8].BIT3	Présence 6-pack zone 3
compteur_S3	Capteur	INT[8].BIT4	Identification de palette zone 3
palette_S4	Capteur	INT[11].BIT1	Présence palette zone 4
sortie_S4	Capteur	INT[11].BIT2	Sortie de la zone 4
pack_S4	Capteur	INT[11].BIT3	Présence 6-pack zone 4
compteur_S4	Capteur	INT[11].BIT4	Identification de palette zone 4
palette_S5	Capteur	INT[14].BIT1	Présence palette zone 5
sortie_S5	Capteur	INT[14].BIT2	Sortie de la zone 5
pack_S5	Capteur	INT[14].BIT3	Présence 6-pack zone 5
compteur_S5	Capteur	INT[14].BIT4	Identification de palette zone 5
palette_S6	Capteur	INT[17].BIT1	Présence palette zone 6
sortie_S6	Capteur	INT[17].BIT2	Sortie de la zone 6
pack_S6	Capteur	INT[17].BIT3	Présence 6-pack zone 6
compteur_S6	Capteur	INT[17].BIT4	Identification de palette station 6
sorti_impexp	Var. Interne	DB2.DBX0.0	Information de l'import/export
haut_impexp	Var. Interne	DB2.DBX0.1	Information de l'import/export
rentre_trans	Var. Interne	DB3.DBX0.0	Information du transfert
haut_trans	Var. Interne	DB3.DBX0.1	Information du transfert
Lumiere_ON_auto	Actionneur	%Q124.0	Voyant « ON » Auto
Lumiere_OFF_auto	Actionneur	%Q124.1	Voyant « OFF » Auto
Convoyeur	Actionneur	%Q124.7	Activation du Convoyeur
Ver_S1	Actionneur	OUT[3].BIT1	Vérin bloqueur station 1
Ver_S2	Actionneur	OUT[6].BIT1	Vérin bloqueur station 2
Ver_S3	Actionneur	OUT[9].BIT1	Vérin bloqueur station 3
Ver_S4	Actionneur	OUT[12].BIT1	Vérin bloqueur station 4
Ver_S5	Actionneur	OUT[15].BIT1	Vérin bloqueur station 5
Ver_S6	Actionneur	OUT[18].BIT1	Vérin bloqueur station 6

cette exigence est présenté ci-dessous :

Ensemble de contraintes pour l'exigence n° 3

$$\begin{array}{lcllcl} CSs_1 & = & palette\_S1 \cdot \downarrow Ver\_S1 & CSs_4 & = & palette\_S4 \cdot \downarrow Ver\_S4 \\ CSs_2 & = & palette\_S2 \cdot \downarrow Ver\_S2 & CSs_5 & = & palette\_S5 \cdot \downarrow Ver\_S5 \\ CSs_3 & = & palette\_S3 \cdot \downarrow Ver\_S3 & CSs_6 & = & palette\_S6 \cdot \downarrow Ver\_S6 \end{array}$$

L'exigence n° 4 se traduit par deux contraintes :

Contrainte pour l'exigence n° 4

$$\begin{array}{lcl} CSs_7 & = & sorti\_impexp \cdot \overline{haut\_impexp} \cdot \uparrow Ver\_S6 \\ CSs_8 & = & \overline{rentre\_trans} \cdot \overline{haut\_trans} \cdot \uparrow Ver\_S2 \end{array}$$

V-2.1.3 Analyse de la cohérence des contraintes de sécurité

L'approche d'analyse de la cohérence développée dans cette thèse a été présentée dans la section III-4.3. Pour rappel, cette approche se base sur la construction de deux graphes. Le premier, appelé graphe structurel, représente les liens structurels entre contraintes et permet de réduire le nombre de contraintes à tester. Le second, appelé graphe d'atteignabilité, représente les affectations successives appliquées par le filtre lors de la résolution des contraintes. L'ensemble de contraintes est cohérent si et seulement si il n'existe pas de cycle dans le graphe d'atteignabilité.

Le graphe structurel pour les contraintes du convoyeur est présenté dans la figure 53. Il n'existe pas de lien entre les contraintes. Par conséquent, lors de la résolution d'une contrainte il ne sera jamais possible de violer une autre contrainte. Les contraintes sont donc **cohérentes** et il n'est pas nécessaire de construire le graphe d'atteignabilité.

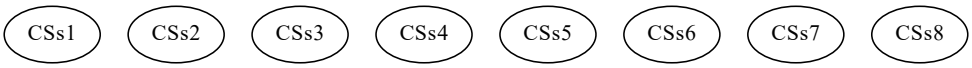


FIGURE 53 – Graphe structurel pour le convoyeur central

V-2.1.4 Implémentation du filtre de sécurité

Le programme ST permettant d'implémenter le filtre de sécurité à partir des contraintes précédentes est présenté ci-après. Ce programme a été généré automatiquement en utilisant le logiciel SEDMA. Étant donné qu'il n'existe pas de contrainte combinée dans l'ensemble de contraintes, la version simple du filtre est utilisée (Alg. 1).

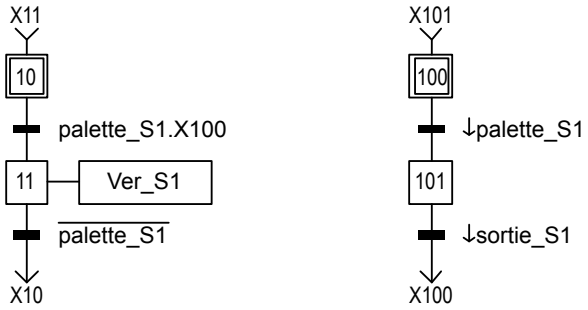
```
(* Affectation des Gk à partir des actionneurs *)
G[0] := Ver_S1;
G[1] := Ver_S2;
G[2] := Ver_S3;
G[3] := Ver_S4;
G[4] := Ver_S5;
G[5] := Ver_S6;

(* Création des F0sk à partir des CSs *)
F0s[0] := F0s[2] := F0s[3] := F0s[4] := false;
F0s[1] := NOT transfert_rentre AND NOT transfert_haut AND NOT memQ1;
F0s[5] := impExp_sorti AND NOT impExp_haut AND NOT memQ5;

(* Création des F1sk à partir des CSs *)
F1s[0] := palette_S1 AND memQ0;
F1s[1] := palette_S2 AND memQ1;
F1s[2] := palette_S3 AND memQ2;
F1s[3] := palette_S4 AND memQ3;
F1s[4] := palette_S5 AND memQ4;
F1s[5] := palette_S6 AND memQ5;

(* Affectation des sorties finales *)
Ver_S1 := NOT F0s[0] AND G[0] OR F1s[0];
Ver_S2 := NOT F0s[1] AND G[1] OR F1s[1];
Ver_S3 := NOT F0s[2] AND G[2] OR F1s[2];
Ver_S4 := NOT F0s[3] AND G[3] OR F1s[3];
Ver_S5 := NOT F0s[4] AND G[4] OR F1s[4];
Ver_S6 := NOT F0s[5] AND G[5] OR F1s[5];

(* Mémorisation *)
memQ0 := Ver_S1;
memQ1 := Ver_S2;
memQ2 := Ver_S3;
memQ3 := Ver_S4;
memQ4 := Ver_S5;
memQ5 := Ver_S6;
```

V-2.1.5 Définition du programme fonctionnel

Pour cette station (convoyeur central), le programme fonctionnel a été réalisé en sachant qu'un filtre de sécurité est présent (filtre correcteur contrôleur (section I-4.3.2)). La réalisation du programme fonctionnel est alors assez simple car il n'est pas nécessaire d'effectuer

toutes les vérifications obligatoires avant une action. En effet, si une action ne doit pas être effectuée le filtre la désactivera jusqu'à ce que celle-ci puisse être autorisée.

L'exigence n° 1 ne nécessite pas de spécification, le convoyeur est activé dès l'allumage de l'automate. Les grafcets permettant la spécification de l'exigence n° 2, pour la zone 1, sont présentés dans la figure 54. Les mêmes types de grafcets existent pour les six zones du convoyeur.



(a) Vérin Ver_S1 (b) Section 1

FIGURE 54 – Grafcets du programme fonctionnel pour la zone 1 du convoyeur central

Le grafcet 54a permet de baisser le vérin 1 lorsqu'une palette est présente au dessus du capteur de présence de la station 1. Le grafcet 54b permet de savoir si une palette est en cours de transit sur la section 1, c'est à dire entre les capteurs $palette\_S1$ et $sortie\_S1$.

Ces grafcets ne permettent pas de garantir les exigences n° 3 et n° 4 du cahier des charges, néanmoins les contraintes précédentes le permettent. Dès lors, en plaçant le programme fonctionnel avant l'exécution du filtre de sécurité dans le cycle automate, l'ensemble des exigences seront respectées par l'API.

L'utilisation du filtre logique, dans ce cas, permet de simplifier la conception du programme fonctionnel.

V-2.2 Transfert

La station de transfert de la cellule flexible est une station permettant de charger (ou décharger) des bouteilles vers (ou depuis) des 6-packs présents sur le convoyeur central.

Elle se compose de deux convoyeurs. Le premier permet de récupérer les bouteilles sortantes de l'embouteillage. Le second permet d'acheminer les bouteilles vers une zone d'at-

tente de transfert ou bien d'amener les bouteilles vers la station bras robotisée (zone 1).

Un vérin bloqueur est disposé en sortie de la zone d'attente afin d'empêcher les bouteilles de continuer vers la station du bras robot.

Le transfert des bouteilles en attente s'effectue à l'aide de deux vérins double effet (commandable en sortie et rentrée) et d'une pince. Ces deux vérins sont attachés bout à bout. Le premier assure le déplacement horizontal (axe X) et le second le déplacement vertical (axe Z). Sur la course du vérin horizontal, est placé un taquet permettant de bloquer physiquement la sortie du vérin dans une position intermédiaire. Ainsi il est possible de déposer les bouteilles côte à côte dans le 6-pack. Le vérin vertical est équipé d'un frein mécanique empêchant tout mouvement lorsqu'il n'est pas relâché.

Enfin, une pince est présente au bout du vérin vertical. Celle-ci est commandable en ouverture, et est étudiée pour attraper les bouteilles trois par trois afin de les placer dans le 6-pack. Il existe également des capteurs détectant la présence d'une, deux, trois, quatre et six bouteilles en attente sur le convoyeur.

La figure 55 propose une vue schématisée de côté de la station de transfert. Seule une partie des capteurs et actionneurs y sont représentés par souci de lisibilité.

La liste des variables utilisées pour cette station est détaillée dans le tableau 18. Les mémoires ne sont pas listées dans ce tableau, pour rappel la mémoire (valeur au cycle précédent) de la variable var est notée var^{-1}.

V-2.2.1 Cahier des charges

Le cahier des charges de cette station est séparé en plusieurs parties présentées ci-après.

Chargement d'un 6-pack Lorsque six bouteilles sont en attente de transfert, le chargement peut s'effectuer lorsqu'un 6-pack vide est présent et en attente sur le convoyeur central dans la zone 2. Les bouteilles doivent être chargées 3 par 3. La première série de bouteilles est placée dans le premier emplacement du 6-pack, la seconde série dans le second emplacement du 6-pack.

TABLE 18 – Mnémoniques pour la station de transfert

Mnémonique	Type	Adresse	Description
ouvert	Capteur	%I6.0	Pince ouverte
ferme	Capteur	%I6.1	Pince fermée
haut	Capteur	%I6.2	Vérin axe Z en position haute
bas	Capteur	%I6.3	Vérin axe Z en position basse
rentre	Capteur	%I7.0	Vérin axe X en position rentrée
sorti	Capteur	%I7.1	Vérin axe Z en position sortie
milieu	Capteur	%I7.2	Vérin axe Z en position milieu
bouteille_1	Capteur	%I8.0	Une bouteille en attente
bouteille_3	Capteur	%I8.2	Trois bouteilles en attente
bouteille_6	Capteur	%I8.3	Six bouteilles en attente
bloqueur_OUT_sorti	Capteur	%I9.0	Bloqueur_OUT sorti
bouteille_2	Capteur	%I9.1	Deux bouteilles en attente
bloqueur_IN_sorti	Capteur	%I9.2	Bloqueur_IN sorti
bouteille_4	Capteur	%I9.3	Quatre bouteilles en attente
Convoyeur	Actionneur	%Q4.0	Activation du convoyeur
Ouverture_Pince	Actionneur	%Q4.1	Ouverture de la pince
Monter	Actionneur	%Q4.2	Rentrer vérin axe Z
Descendre	Actionneur	%Q4.3	Sortir vérin axe Z
Rentrer	Actionneur	%Q5.0	Rentrer vérin axe X
Sortir	Actionneur	%Q5.1	Sortir vérin axe X
Pos_Inter	Actionneur	%Q5.2	Bloqueur de position intermédiaire
Rel_Frein	Actionneur	%Q5.3	Relâcher le frein vertical
Bloqueur_OUT	Actionneur	%Q7.0	Bloque la sortie des bouteilles
Bloqueur_IN	Actionneur	%Q7.2	Bloque l'entrée des bouteilles
bouteilles_serrees	Observateur	%M1.0	Présence de bouteilles dans la pince
presenceBouteille_4_3	Observateur	%M3.2	Bouteille devant Bloqueur_IN
palette_S2	Var. Interne	%M3.4	Information partagée
pack_S2	Var. Interne	%M3.5	Information partagée
Ver_S2	Var. Interne	%M3.6	Information partagée

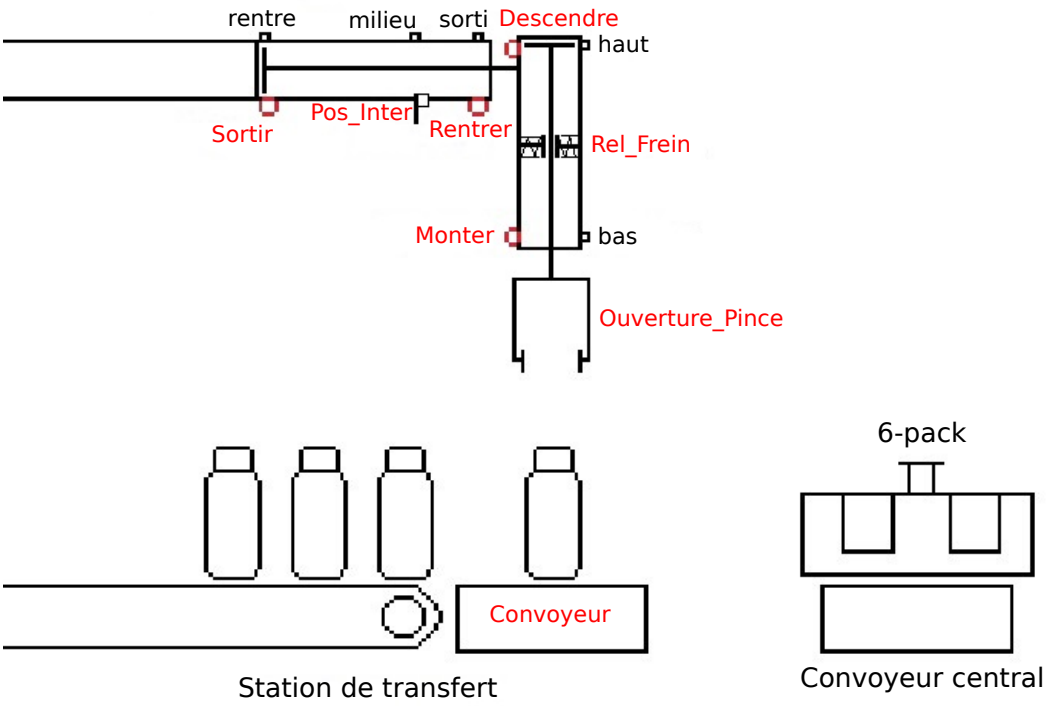


FIGURE 55 – Description de la station de transfert

Déchargement d'un 6-pack Lorsqu'une demande de déchargement est effectuée et qu'un 6-pack plein est présent et en attente sur le convoyeur central dans la zone 2, les bouteilles doivent être déchargées 3 par 3. Puis ces bouteilles doivent être acheminées vers la station du bras robot pour être dé-bouchonnées et vidées de leur contenant.

Sécurité Les collisions doivent être évitées. Par exemple, il n'est pas possible de descendre la pince fermée si des bouteilles sont présentes en dessous. D'un point de vue général, tout mouvement dangereux doit être interdit.

Convoyeurs Les deux convoyeurs sont reliés sur la même sortie automate. Le premier permet d'acheminer les bouteilles provenant de la station d'embouteillage vers la station de transfert. Le second convoyeur permet d'acheminer les bouteilles vers la zone d'attente en dessous de la pince ou alors d'acheminer les bouteilles vers la station du bras robot.

V-2.2.2 Définition des contraintes de sécurité

La description du cahier des charges pour la partie sécurité est très succincte. Ces exigences ne détaillent pas l'ensemble des cas à interdire, il est donc nécessaire d'établir une analyse plus approfondie des différentes situations dangereuses. Dans cette partie les différentes exigences supplémentaires devant être ajoutées afin de garantir la sécurité sont présentées. Les contraintes de sécurité permettant de spécifier ces exigences sont ensuite détaillées.

Pour cela, les exigences et contraintes sont regroupées en différents groupes en fonction du mouvement et/ou des actionneurs considérés.

Déplacement horizontal (axe X) Il est dangereux de rentrer où de sortir le vérin horizontal dans le cas où la pince ne serait pas en haut :

$$\begin{aligned} CSs_1 &= \overline{haut} \cdot Sortir \\ CSs_2 &= \overline{haut} \cdot Rentrer \end{aligned}$$

Les vérins et pré-actionneurs utilisés pour cet axe impliquent qu'un mouvement complet du vérin a lieu même lorsque l'ordre n'est pas maintenu. Par conséquent, il n'est pas possible de se retrouver dans une situation pouvant poser problème. Aucune autre contrainte n'est alors nécessaire pour garantir le bon fonctionnement de cet axe.

Déplacement vertical (axe Z) Le vérin de cet axe a le même comportement que le précédent. Néanmoins, un frein existe et lorsque celui-ci n'est pas activé aucun mouvement n'est effectué.

Pour éviter des efforts mécaniques inutiles sur le frein, celui-ci ne doit pas être désactivé lorsque le vérin vertical est en mouvement :

$$CSs_3 = \overline{haut} \cdot \overline{bas} \cdot \overline{Rel\_Frein}$$

Le bras ne doit pas descendre si des bouteilles sont présentes en dessous et que la pince n'est pas ouverte :

$$\begin{aligned} CSs_4 &= haut \cdot rentre \cdot bouteille\_1 \cdot \overline{ouvert} \cdot Descendre \\ CSs_5 &= haut \cdot rentre \cdot bouteille\_2 \cdot \overline{ouvert} \cdot Descendre \\ CSs_6 &= haut \cdot rentre \cdot bouteille\_3 \cdot \overline{ouvert} \cdot Descendre \end{aligned}$$

La zone de prise/dépose des bouteilles doit être fermée avant de descendre le bras pour éviter les collisions avec les bouteilles :

$$CSs_7 = rentre \cdot Convoyeur^{-1} \cdot \overline{bloqueur\_IN\_sorti} \cdot Descendre$$
$$CSs_8 = rentre \cdot Convoyeur^{-1} \cdot \overline{bloqueur\_OUT\_sorti} \cdot Descendre$$

La descente au dessus du convoyeur central ne doit s'effectuer que lorsqu'un 6-pack est présent et que le vérin bloqueur correspondant (Ver_S2) n'est pas descendu :

$$CSs_9 = haut \cdot \overline{rentre} \cdot \overline{pack\_S2} \cdot Descendre$$
$$CSs_{10} = haut \cdot \overline{rentre} \cdot Ver\_S2 \cdot Descendre$$

Afin d'éviter des mouvements violents du vérin, une procédure doit être suivie pour autoriser un mouvement. En effet, en position haute, l'ordre de Descendre ne doit pouvoir être activé que si l'ordre de Monter était activé préalablement. Le mouvement de descente s'effectue alors par un front descendant de l'ordre de Monter tout en maintenant l'ordre de Descendre. Le même type de règle s'applique au mouvement opposé (Monter). Les contraintes nécessaires pour obliger ce fonctionnement, et ainsi garantir la sécurité, sont les suivantes :

$$CSs_{11} = haut \cdot \overline{Monter^{-1}} \cdot \uparrow Descendre$$
$$CSs_{12} = bas \cdot \overline{Descendre^{-1}} \cdot \uparrow Monter$$

De plus, le frein ne doit pas être désactivé si les mauvais ordres sont enclenchés. Les deux contraintes combinées suivantes, couplées aux deux contraintes simples précédentes, permettent d'éviter ce genre de situation. De plus, la priorité fonctionnelle est donnée au frein. Par conséquent, les ordres de mouvements doivent être désactivés.

$$CSc_1 = haut \cdot \overline{Rel\_Frein} \cdot \mathbf{Descendre}$$
$$CSc_2 = bas \cdot \overline{Rel\_Frein} \cdot \mathbf{Monter}$$

Dans les contraintes combinées, les variables en **gras** indiquent les variables forcées pour traduire la priorité fonctionnelle de cette contrainte.

Bloqueurs de bouteilles Deux petits vérins sont présents au niveau du convoyeur de la station. Ces vérins permettent de bloquer l'accès des bouteilles à la zone se situant sous le bras. Ce sont des vérins simples effets monostables, l'annulation de l'ordre implique la rentrée automatique des vérins.

Les vérins ne doivent pas pouvoir rentrer en collision avec des bouteilles. Dû à un écart

important entre les capteurs de présence 3 et 4, un observateur est créé afin de savoir si une bouteille est présente entre ces deux capteurs :

$$presenceBouteille\_4\_3 \quad = \quad \uparrow bouteille\_4 + presenceBouteille\_4\_3^{-1} \cdot \overline{\uparrow bouteille\_3}$$

Les contraintes suivantes sont alors possibles :

$$\begin{aligned} CSs_{13} &= presenceBouteille\_4\_3 \cdot \uparrow Bloqueur\_IN \\ CSs_{14} &= Convoyeur^{-1} \cdot bouteille\_1 \cdot \uparrow Bloqueur\_OUT \\ CSs_{15} &= Convoyeur^{-1} \cdot bouteille\_4 \cdot \uparrow Bloqueur\_IN \end{aligned}$$

Enfin, lors de la dépose de bouteilles entre les vérins bloqueurs, le bloqueur de sorti doit rester sorti tant que la dépose n'est pas terminée :

$$\begin{aligned} CSs_{16} &= Convoyeur^{-1} \cdot \overline{haut} \cdot rentre \cdot bouteille\_1 \cdot ouvert \cdot \downarrow Bloqueur\_OUT \\ CSs_{17} &= Convoyeur^{-1} \cdot \overline{haut} \cdot rentre \cdot bouteille\_2 \cdot ouvert \cdot \downarrow Bloqueur\_OUT \\ CSs_{18} &= Convoyeur^{-1} \cdot \overline{haut} \cdot rentre \cdot bouteille\_3 \cdot ouvert \cdot \downarrow Bloqueur\_OUT \end{aligned}$$

Convoyeur d'acheminement de bouteilles Le convoyeur ne doit pas être activé si la pince risque d'être en position basse au dessus de la zone d'attente (CSs_{17}). En effet, cela pourrait engendrer une collision avec des bouteilles présentes sur le convoyeur. De plus, si des bouteilles sont présentes sur le convoyeur dans la pince ouverte, le convoyeur ne doit pas être activé si le bloqueur de sorti n'est pas activé. Dans le cas contraire, le convoyeur entraînerait les bouteilles qui risqueraient de se retrouver coincées par la pince.

$$\begin{aligned} CSs_{19} &= \overline{haut} \cdot rentre \cdot Convoyeur \\ CSs_{20} &= \overline{haut} \cdot rentre \cdot ouvert \cdot bouteille\_1 \cdot \overline{bloqueur\_OUT\_sorti} \cdot Convoyeur \\ CSs_{21} &= \overline{haut} \cdot rentre \cdot ouvert \cdot bouteille\_2 \cdot \overline{bloqueur\_OUT\_sorti} \cdot Convoyeur \\ CSs_{22} &= \overline{haut} \cdot rentre \cdot ouvert \cdot bouteille\_3 \cdot \overline{bloqueur\_OUT\_sorti} \cdot Convoyeur \end{aligned}$$

Ouverture de la pince Pour la gestion de la pince, la situation dangereuse est l'ouverture de la pince lorsqu'elle serre des bouteilles et n'est pas en position basse au dessus du convoyeur ou d'un 6-pack. Aucun capteur n'étant présent pour indiquer si la pince serre des bouteilles, un observateur logique *bouteilles_serrees* est créé. Ce dernier est ensuite utilisé pour empêcher l'ouverture de la pince :

$$\begin{aligned} bouteilles\_serrees &= bas \cdot \downarrow ouvert + bouteilles\_serrees^{-1} \cdot \overline{\downarrow ferme} \\ CSs_{23} &= \overline{bas} \cdot bouteilles\_serrees \cdot Ouverture\_Pince \end{aligned}$$

La pince ne doit pas changer d'état lors d'un déplacement :

$$\begin{aligned} CSs_{24} &= \overline{haut} \cdot \overline{bas} \cdot \uparrow Ouverture\_Pince \\ CSs_{25} &= \overline{haut} \cdot \overline{bas} \cdot \downarrow Ouverture\_Pince \end{aligned}$$

Mouvements simultanés Il est dangereux d'autoriser des mouvements simultanés. Par exemple, monter et sortir simultanément peut impliquer des chocs avec les parties fixes du système. Pour ce genre de mouvements diagonaux, aucune priorité n'est donnée et les deux ordres concernés sont désactivés :

$$\begin{aligned} CSc_3 &= \textbf{Monter} \cdot \textbf{Sortir} & CSc_5 &= \textbf{Descendre} \cdot \textbf{Rentrer} \\ CSc_4 &= \textbf{Monter} \cdot \textbf{Rentrer} & CSc_6 &= \textbf{Descendre} \cdot \textbf{Sortir} \end{aligned}$$

Contraintes structurelles Les contraintes structurelles sont utilisées pour l'analyse de la cohérence des contraintes. Celles-ci permettent la suppression de vecteurs impossibles à obtenir sur le système réel (considéré sans défaut).

Contraintes structurelles sur les capteurs

$$\begin{aligned} struct_1 &= rentre \cdot sorti & struct_4 &= haut \cdot bas \\ struct_2 &= rentre \cdot milieu & struct_5 &= ouvert \cdot ferme \\ struct_3 &= milieu \cdot sorti \end{aligned}$$

Contraintes structurelles d'état initiale

$$\begin{aligned} struct_6 &= Monter^{-1} \cdot Sortir^{-1} & struct_8 &= Descendre^{-1} \cdot Rentrer^{-1} \\ struct_7 &= Monter^{-1} \cdot Rentrer^{-1} & struct_9 &= Descendre^{-1} \cdot Sortir^{-1} \end{aligned}$$

V-2.2.3 Analyse de la cohérence des contraintes de sécurité

L'analyse de la cohérence de ces contraintes a été réalisée avec le logiciel SEDMA. Tout comme pour le convoyeur central, le graphe structurel ne contient aucun lien entre les différentes contraintes. Par conséquent, les contraintes de sécurité de la station de transfert sont **cohérentes**.

V-2.2.4 Implémentation du filtre de sécurité

L'ensemble de contraintes contient des contraintes combinées et des contraintes simples. Par conséquent, la version « complète » de l'algorithme du filtre est utilisée (Alg. 2).

L'implémentation de cet algorithme en langage ST a été généré automatiquement avec le logiciel SEDMA. Néanmoins, par soucis de lisibilité le programme n'est pas présenté.

V-2.2.5 Définition du programme fonctionnel

Pour cette station de transfert, à la place d'un programme réalisant les actions de manière automatique, un mode manuel et une interface homme machine (IHM) ont été réalisé pour permettre l'activation et la désactivation des actionneurs par un opérateur humain (Fig. 56). Les états des différents capteurs sont également indiqués.

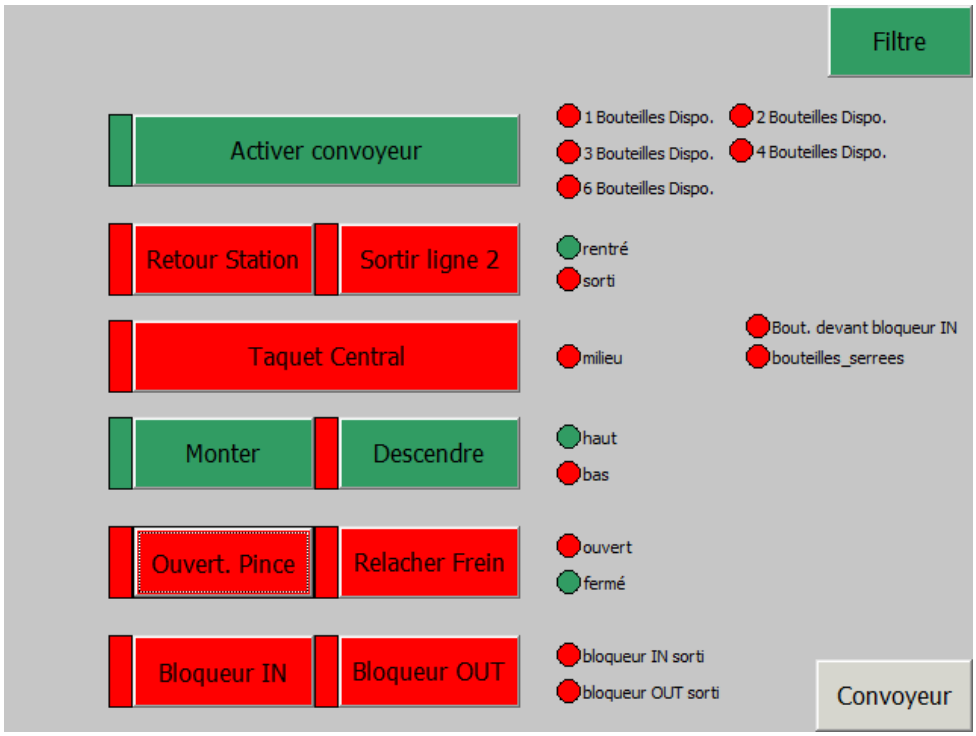


FIGURE 56 – IHM de contrôle manuel pour la station de transfert

Il est possible d'activer ou de désactiver le filtre logique afin de se rendre compte des fonctionnements interdits par le filtre. Lorsque le filtre est activé, une demande d'activation d'un actionneur par le pupitre n'entraînera donc pas obligatoirement son activation. Un indicateur est donc présent à côté du bouton d'activation afin de connaître l'état réel de

l'actionneur. Par exemple, dans le cas présenté dans la figure 56, la CSs_7 et la CSc_1 force l'ordre *Descendre* à $false$ alors que le pupitre demande son activation.

Dans un contexte de formation, cette utilisation du filtre logique permet la prise en main du système sans risque de détérioration de celui-ci. On notera que le filtre correcteur « contrôleur » permet ici d'implémenter simplement un mode de marche manuel.

V-2.3 Import/Export

La station d'import/export est très similaire à celle du transfert de bouteilles traitée précédemment. La principale différence entre les deux est l'ajout d'un troisième axe permettant un déplacement latéral. De plus, le produit transporté n'est pas des bouteilles mais des 6-packs. Ces 6-packs peuvent être pleins ou vides.

La station consiste en : un vérin double-effet pour chaque axe de déplacement (X, Y, Z), un convoyeur permettant l'arrivée de nouveaux 6-packs, deux lignes inclinées permettant l'exportation de 6-packs, des petits vérins assurant la stabilité durant la pose d'un 6-pack sur les lignes inclinées, un petit vérin bloqueur central permettant de positionner le système au-dessus de la ligne d'exportation 1, et pour finir d'une pince (accrochée au vérin vertical) permettant d'agripper les 6-packs.

Tout comme dans la station précédente, le vérin vertical est équipé d'un frein. Celui-ci empêchant tout déplacement lorsqu'il n'est pas relâché.

Une vue schématique du dessus est disponible sur la figure 57.

La liste des variables utilisées pour cette station est détaillée dans le tableau 19. Les observateurs sont détaillés par la suite de cette section.

V-2.3.1 Cahier des charges

Le cahier des charges de cette station est séparé en plusieurs parties présentées ci-après.

Import d'un 6-pack L'importation consiste à insérer, sur le convoyeur central, un nouveau 6-pack (en général vide). Lorsqu'un 6-pack est présent en entrée du convoyeur

TABLE 19 – Mnémoniques pour l'import/export

Mnémonique	Type	Adresse	Description
rentre	Capteur	%I0.0	Vérin axe Y en position rentrée
sorti	Capteur	%I0.1	Vérin axe Y en position sortie
bas	Capteur	%I0.2	Vérin axe Z en position sortie
haut	Capteur	%I0.3	Vérin axe Z en position rentrée
ferme	Capteur	%I1.0	Pince en position fermée
ouvert	Capteur	%I1.1	Pince en position ouverte
l2_pleine	Capteur	%I1.2	Ligne d'extraction 2 pleine
vl2_sorti	Capteur	%I1.3	Vérin d'amortissage de pose ligne 2 sorti
p_gauche	Capteur	%I2.0	Vérin axe X en position sortie
p_droite	Capteur	%I2.1	Vérin axe X en position rentrée
p_milieu	Capteur	%I2.2	Vérin axe X en position intermédiaire
stopm_b	Capteur	%I2.3	Bloqueur central axe X rentré
stopm_h	Capteur	%I3.0	Bloqueur central axe X sorti
l1_occupe	Capteur	%I3.2	Ligne d'extraction 1 en utilisation
l2_occupe	Capteur	%I3.3	Ligne d'extraction 2 en utilisation
bac_pret	Capteur	%I4.0	Présence d'un 6-pack en fin de convoyeur
bac_occupe	Capteur	%I4.1	Présence d'un 6-pack en début de convoyeur
l1_pleine	Capteur	%I4.2	Ligne d'extraction 1 pleine
vl1_sorti	Capteur	%I4.3	Vérin d'amortissage de pose ligne 1 sorti
pbac	Observateur	%M13.5	6-pack présent dans la pince
odroite	Observateur	%M13.6	Origine de l'axe X avant mouvement
ogauche	Observateur	%M13.7	Origine de l'axe X avant mouvement
deposel1	Observateur	%M14.1	Vérin de la ligne 1 sorti
deposel2	Observateur	%M14.2	Vérin de la ligne 2 sorti
Ver_S6	Var. interne	%M124.0	Valeur partagée du vérin 6 du convoyeur central
FPince	Actionneur	%Q0.0	Fermeture de la Pince
OPince	Actionneur	%Q0.1	Ouverture de la Pince
Descente	Actionneur	%Q0.2	Sortie du vérin axe Z
Monter	Actionneur	%Q0.3	Rentrée du vérin axe Z
Rel_Frein	Actionneur	%Q1.0	Relâche du frein vertical
Rentrer	Actionneur	%Q1.2	Rentrée du vérin axe Y
Sortir	Actionneur	%Q1.3	Sortie du vérin axe Y
Gauche	Actionneur	%Q2.0	Sortie du vérin axe X
Droite	Actionneur	%Q2.1	Rentrée du vérin axe X
Stop_M	Actionneur	%Q2.2	Sortie du bloqueur central axe X
Sortie_l1	Actionneur	%Q3.0	Sortie du vérin de la ligne 1
Sortie_l2	Actionneur	%Q3.2	Sortie du vérin de la ligne 2
Conv	Actionneur	%Q4.0	Activation du convoyeur

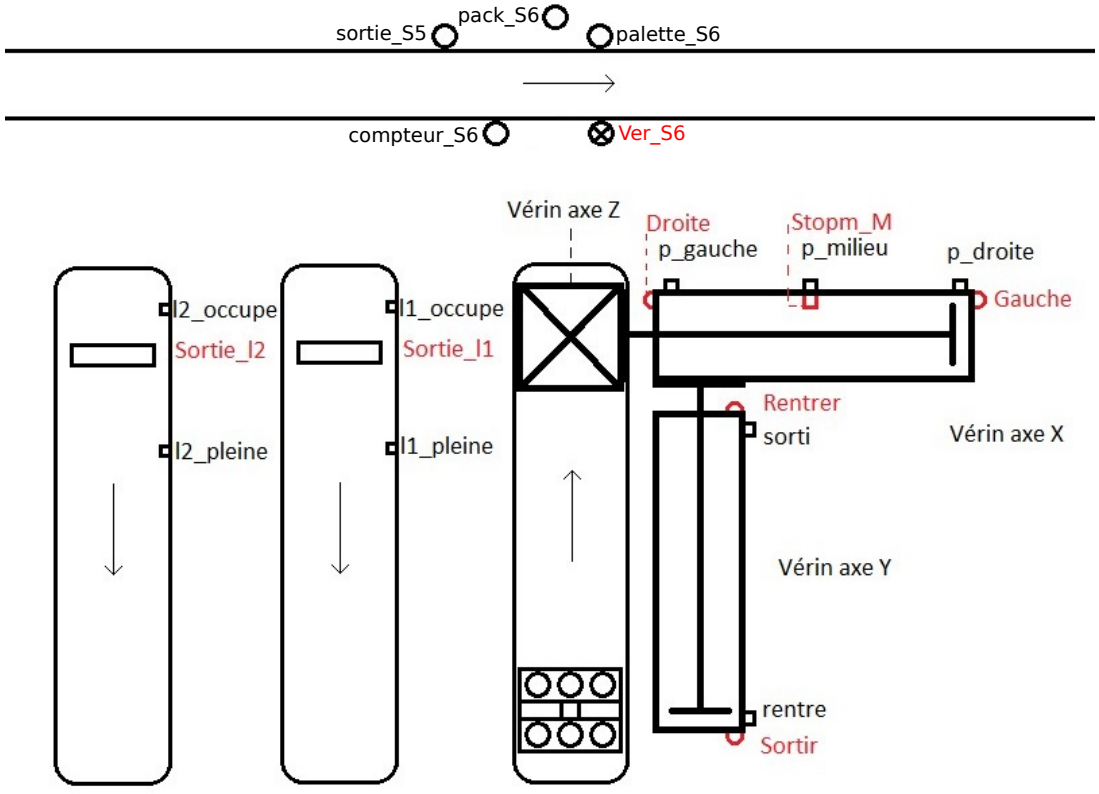


FIGURE 57 – Description de la station d'import/export

d'importation, il doit être acheminé à la fin du convoyeur, en dessous du vérin vertical. Lorsqu'une palette vide se présente à la bonne position sur le convoyeur centrale (capteur *palette_S6*), le 6-pack en attente est chargé sur la palette.

Export d'un 6-pack L'exportation consiste à prendre depuis le convoyeur central un 6-pack (en général plein). Lorsqu'un 6-pack est présent sur le convoyeur central, face à la station, les vérins doivent être mis en mouvement afin de prendre le 6-pack, puis le déposer sur une des deux lignes d'exportation. L'exportation est prioritaire sur l'importation, ceci afin d'éviter un blocage du système.

Gestion des lignes d'évacuation Il existe 2 lignes d'exportation, celles-ci sont inclinées permettant d'acheminer les 6-packs facilement vers la sortie de la station. La ligne 1 doit être remplie en priorité, ceci afin d'optimiser le temps d'exportation. Lorsque la ligne 1 est pleine, la ligne 2 doit être utilisée pour l'exportation.

Sécurité La sécurité doit être garantie sur ce système. Les principaux risques sont liés à des collisions entre palettes, ou bien entre les vérins et les palettes. La gestion du frein sur le vérin vertical est également considéré comme un élément de sécurité. En effet, si ce frein est mal géré, des détériorations du système peuvent être provoquées. Enfin, certains mouvements doivent être interdits, par exemple : déposer un 6-pack sur le convoyeur central alors qu'il n'y a pas de palette.

V-2.3.2 Définition des contraintes de sécurité

Cette partie a pour objectif de décrire pas à pas l'ensemble des contraintes. Dans un premier temps, cinq observateurs logiques sont définis pour faciliter l'écriture des contraintes.

observateurs logiques Il n'existe pas de capteur permettant de savoir si un 6-pack est présent dans la pince. Afin d'avoir accès à cette information, l'observateur *pbac* est défini.

$$pbac \quad = \quad bas \cdot \uparrow ferme + pbac^{-1} \cdot \overline{ouvert}$$

Au niveau des lignes d'exportation 1 et 2, il n'existe pas de capteur de fin de course pour les vérins de stabilisation. Un observateur est alors nécessaire afin d'être certain que les vérins sont effectivement sortis.

$$\begin{aligned} deposel1 &= bas \cdot p\_milieu \cdot vl1\_sorti + deposel1^{-1} \cdot \overline{haut} \\ deposel2 &= bas \cdot p\_gauche \cdot vl2\_sorti + deposel2^{-1} \cdot \overline{haut} \end{aligned}$$

Enfin, pour la gestion de l'axe X, il est nécessaire de savoir de quel côté le déplacement a commencé. Cette information sera utilisée pour la gestion du déplacement lorsque le vérin servant de taquet au milieu du déplacement sera utilisé.

$$\begin{aligned} odroite &= \downarrow p\_droite + odroite^{-1} \cdot \overline{ogauche} \\ ogauche &= \downarrow p\_gauche + ogauche^{-1} \cdot \overline{odroite} \end{aligned}$$

Déplacement horizontal (axe X) L'axe X est un vérin double-effet. Il permet le déplacement au dessus des trois lignes (exportation et importation des 6-packs). Afin de réduire l'usure du système et d'éviter une casse, il est nécessaire d'empêcher un maintien d'ordre (Gauche ou Droite) si l'axe est arrivé en butée. De plus, cet axe comporte une particularité. En effet, un petit vérin simple effet est placé sur sa course. Si ce vérin

est activé, il va venir empêcher le déplacement de l'axe X afin de positionner celui-ci au dessus de la ligne d'exportation n° 1. Enfin, le déplacement est interdit si la pince n'est pas en position haute, dans le cas contraire un risque de collision existe. Les contraintes de sécurité pour cet axe sont présentées ci-dessous :

Empêcher le maintien d'ordre si axe en butée

$$
\begin{aligned}
CSs_1 &= \overline{p\_droite} \cdot \overline{p\_milieu} \cdot \uparrow Gauche \\
CSs_2 &= \overline{p\_gauche} \cdot \overline{p\_milieu} \cdot \uparrow Droite \\
CSs_3 &= p\_droite \cdot Droite \\
CSs_4 &= p\_gauche \cdot Gauche \\
CSs_5 &= p\_milieu \cdot stopm\_h \cdot odroite \cdot Gauche \\
CSs_6 &= p\_milieu \cdot stopm\_h \cdot ogauche \cdot Droite
\end{aligned}
$$

Maintien du mouvement en cours

$$
\begin{aligned}
CSs_7 &= \overline{p\_droite} \cdot \overline{stopm\_h} \cdot \downarrow Droite \\
CSs_8 &= \overline{p\_droite} \cdot \overline{p\_milieu} \cdot \downarrow Droite \\
CSs_9 &= \overline{p\_gauche} \cdot \overline{stopm\_h} \cdot \downarrow Gauche \\
CSs_{10} &= \overline{p\_gauche} \cdot \overline{p\_milieu} \cdot \downarrow Gauche
\end{aligned}
$$

Déplacement si position haute

$$
\begin{aligned}
CSs_{11} &= \overline{haut} \cdot \uparrow Gauche \\
CSs_{12} &= \overline{haut} \cdot \uparrow Droite
\end{aligned}
$$

Déplacement horizontal (axe Y) L'axe Y est également un vérin double-effet qui permet le déplacement entre la station d'import/export et le convoyeur central. Les mêmes types de contraintes que pour l'axe X sont donc nécessaires. Néanmoins, celles-ci sont plus simples car il n'y a pas de vérin bloqueur sur la course.

Empêcher le maintien d'ordre si axe en buté

$$
\begin{aligned}
CSs_{13} &= \overline{rentre} \cdot \uparrow Sortir \quad & CSs_{15} &= sorti \cdot Sortir \\
CSs_{14} &= \overline{sorti} \cdot \uparrow Rentrer \quad & CSs_{16} &= rentre \cdot Rentrer
\end{aligned}
$$

Maintien du mouvement en cours

$$
\begin{aligned}
CSs_{17} &= \overline{rentre} \cdot \downarrow Rentrer \\
CSs_{18} &= \overline{sorti} \cdot \downarrow Sortir
\end{aligned}
$$

Déplacement si position haute

$$
\begin{aligned}
CSs_{19} &= \overline{haut} \cdot \uparrow Sortir \\
CSs_{20} &= \overline{haut} \cdot \uparrow Rentrer
\end{aligned}
$$

Déplacement vertical (axe Z) L'axe Z est un vérin double-effet équipé d'un frein. La gestion du frein n'est pas la même que pour celui de la station de transfert. En effet, les pré-actionneurs n'agissent pas de la même façon, il est donc nécessaire de faire quelques

ajustements pour le bon fonctionnement de ce vérin. Par exemple, si aucun ordre de déplacement n'est donné, un maintien de l'ordre précédent doit être effectué. Si cela n'est pas le cas, plusieurs vibrations apparaissent lors du relâchement du frein, ce qui engendre des à-coups importants endommageant le vérin et la structure.

Maintien du mouvement en cours

$$CSs_{21} = \overline{haut} \cdot \downarrow Monter$$
$$CSs_{22} = \overline{Descente^{-1}} \cdot \downarrow Monter$$
$$CSs_{23} = \overline{bas} \cdot \downarrow Descente$$

Maintien d'ordre

$$CSs_{24} = bas \cdot \overline{Monter^{-1}} \cdot \overline{Descente^{-1}} \cdot \overline{Descente}$$
$$CSs_{25} = haut \cdot \overline{Monter^{-1}} \cdot \overline{Descente^{-1}} \cdot \overline{Monter}$$

Pas de frein pendant le mouvement

$$CSs_{26} = \overline{bas} \cdot \overline{haut} \cdot \overline{Rel\_Frein}$$

Gestion des ordres obligatoires en fonction du frein et de la position

$$CSs_{27} = Rel\_Frein^{-1} \cdot haut \cdot \overline{Monter^{-1}} \cdot \overline{Descente^{-1}} \cdot \overline{Monter}$$
$$CSs_{28} = Rel\_Frein^{-1} \cdot \overline{haut} \cdot Monter^{-1} \cdot Descente$$
$$CSs_{29} = Rel\_Frein^{-1} \cdot \overline{bas} \cdot Descente^{-1} \cdot Monter$$
$$CSs_{30} = \overline{Rel\_Frein^{-1}} \cdot \overline{bas} \cdot Descente^{-1} \cdot \uparrow Monter$$
$$CSs_{31} = \overline{Rel\_Frein^{-1}} \cdot \uparrow Descente$$

Conditions d'interdiction de descente

$$CSs_{32} = l1\_occupe \cdot ferme \cdot p\_milieu \cdot rentre \cdot \uparrow Descente$$
$$CSs_{33} = l2\_occupe \cdot ferme \cdot p\_gauche \cdot rentre \cdot \uparrow Descente$$
$$CSs_{34} = \overline{p\_droite} \cdot sorti \cdot \uparrow Descente$$
$$CSs_{35} = p\_droite \cdot sorti \cdot Ver\_S6 \cdot \uparrow Descente$$
$$CSs_{36} = p\_droite \cdot sorti \cdot ferme \cdot pack\_S6 \cdot \uparrow Descente$$
$$CSs_{37} = p\_droite \cdot sorti \cdot \overline{palette\_S6} \cdot \uparrow Descente$$
$$CSs_{38} = p\_droite \cdot sorti \cdot ouvert \cdot \overline{pack\_S6} \cdot \uparrow Descente$$
$$CSs_{39} = \overline{bas} \cdot ferme \cdot p\_droite \cdot rentre \cdot bac\_pret \cdot \uparrow Descente$$
$$CSs_{40} = \overline{bas} \cdot ferme \cdot p\_droite \cdot rentre \cdot Conv^{-1} \cdot \uparrow Descente$$

Gestion des vérins simple effet Les 2 vérins des lignes d'export et le vérin de milieu de course de l'axe X sont de type simple effet.

Vérins des lignes d'évacuation 1 et 2

$$CSs_{41} = \overline{vl1\_sorti} \cdot \downarrow Sortie\_l1 \quad CSs_{43} = \overline{vl2\_sorti} \cdot \downarrow Sortie\_l2$$
$$CSs_{42} = deposel1 \cdot \downarrow Sortie\_l1 \quad CSs_{44} = deposel2 \cdot \downarrow Sortie\_l2$$

Bloqueur central de l'axe X

$$CSs_{45} = \overline{stopm\_b} \cdot \uparrow Stop\_M$$
$$CSs_{46} = \overline{stopm\_h} \cdot \downarrow Stop\_M$$
$$CSs_{47} = \overline{p\_gauche} \cdot \overline{p\_droite} \cdot \uparrow Stop\_M$$

Gestion de la pince La pince est accrochée au bout de l'axe vertical (Z). Si un 6-pack est présent dans la pince (observateur *pbac*), alors la pince ne doit pouvoir s'ouvrir que lorsqu'elle est en bas. De plus, si les vérins stabilisateurs des lignes d'exportation ne sont pas sortis, il ne faut pas lâcher le 6-pack.

Ouverture de la pince

$$CSs_{48} = pbac \cdot \overline{bas} \cdot OPince$$
$$CSs_{49} = \overline{vl1\_sorti} \cdot p\_milieu \cdot pbac \cdot bas \cdot \uparrow OPince$$
$$CSs_{50} = \overline{vl2\_sorti} \cdot p\_gauche \cdot pbac \cdot bas \cdot \uparrow OPince$$

Convoyeur d'import de 6-packs En fin de course du convoyeur est placée une butée mécanique empêchant les 6-packs de tomber. De plus, le système est étudié pour que le 6-pack soit correctement placé en dessous de la pince lorsque celui-ci est en butée. Par contre, le convoyeur ne peut pas être actif si la pince n'est pas en position haute, cela pourrait engendrer une collision.

Arrêt du tapis si nécessaire

$$CSs_{51} = \overline{haut} \cdot p\_droite \cdot Conv$$

Contraintes combinées Sur cette station, les contraintes combinées permettent de garantir l'exclusivité des actions opposées. D'après le cahier des charges, aucune priorité n'existe entre les actionneurs. Les priorités fonctionnelles sont donc choisies de façon arbitraire et guidée par le fonctionnement normal du système. Les priorités fonctionnelles choisies pour cet exemple permettent de favoriser un retour à l'état initial classique du

système : rentré à droite avec la pince fermée. Les variables en **gras** sont celles forcées par les priorités fonctionnelles.

Exclusion mutuelles d'actionneurs

$$CSc_1 = \textbf{Gauche} \cdot Droite$$
$$CSc_2 = \textbf{Sortir} \cdot Rentrer$$
$$CSc_3 = \textbf{OPince} \cdot FPince$$

Gestion de la priorité entre *Descente* et *Monter*

$$CSc_4 = bas \cdot \overline{Monter} \cdot \downarrow \textbf{Descente}$$
$$CSc_5 = bas \cdot \textbf{Monter} \cdot Descente$$

Contraintes structurelles Les contraintes structurelles sont utilisées pour l'analyse de la cohérence des contraintes. Celles-ci permettent la suppression de vecteurs impossibles à obtenir sur le système réel (considéré sans défauts).

Contraintes structurelles sur les capteurs

$$struct_1 = rentre \cdot sorti \qquad struct_5 = p\_gauche \cdot p\_milieu$$
$$struct_2 = haut \cdot bas \qquad struct_6 = p\_milieu \cdot p\_droite$$
$$struct_3 = ouvert \cdot ferme \qquad struct_7 = stopm\_b \cdot stropm\_h$$
$$struct_4 = p\_gauche \cdot p\_droite \qquad struct_8 = odroite \cdot ogauche$$

Contraintes structurelles d'hypothèse initiale

$$struct_9 = Gauche^{-1} \cdot Droite^{-1} \qquad struct_{11} = OPince^{-1} \cdot FPince^{-1}$$
$$struct_{10} = Sortir^{-1} \cdot Rentrer^{-1} \qquad struct_{12} = bas \cdot Monter^{-1} \cdot Descente^{-1}$$

V-2.3.3 Analyse de la cohérence des contraintes de sécurité

Le graphe structurel pour cette station est présenté dans la figure 58. Les contraintes isolées ne sont pas représentées par soucis de lisibilité.

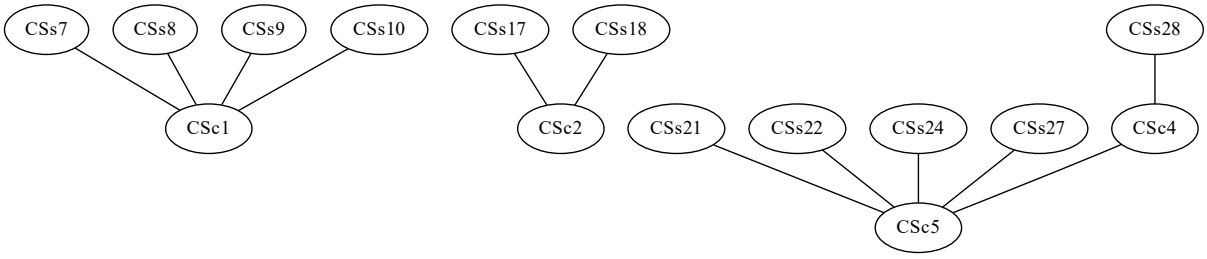


FIGURE 58 – Graphe structurel pour l'import/export

3 groupes de contraintes distincts existent dans ce graphe :

— Groupe 1 : $\{CSc_1, CSs_7, CSs_8, CSs_9, CSs_{10}\}$

— Groupe 2 : $\{CSc_2, CSs_{17}, CSs_{18}\}$

— Groupe 3 : $\{CSc_4, CSc_5, CSs_{21}, CSs_{22}, CSs_{24}, CSs_{27}, CSs_{28}\}$

Pour simplifier la suite de l'analyse, 3 graphes d'atteignabilité différents sont construits et analysés. Le tableau 20 présente les résultats liés à la construction et l'analyse des 3 graphes d'atteignabilité. Trois types de vecteurs sont identifiés : les valides ne violent aucune contrainte, les invalides violent au moins une contrainte, les « struct » sont supprimés du graphe pendant la création car ils ne respectent pas au moins une contrainte structurelle. Les calculs ont été effectués avec le logiciel SEDMA sur un ordinateur portable (i5-5200U, 2.20GHz, 8Go de ram, pas de SSD).

TABLE 20 – Récapitulatif de l'analyse des contraintes de l'import/export

	Groupe 1	Groupe 2	Groupe 3
Variables	9	6	7
Vecteurs valides	76	19	56
Vecteurs invalides	68	17	32
Vecteurs « struct »	368	28	40
Temps de création	308 ms	45 ms	215 ms
Temps d'analyse	19 ms	1 ms	15 ms
Cohérent	oui	oui	oui

V-2.3.4 Implémentation du filtre de sécurité

L'ensemble de contraintes contient des contraintes combinées et des contraintes simples. Par conséquent, la version « complète » de l'algorithme du filtre est utilisée (Alg. 2).

L'implémentation de cet algorithme en langage ST a été généré automatiquement avec le logiciel SEDMA. Néanmoins, par soucis de lisibilité le programme n'est pas présenté.

V-2.3.5 Définition du programme fonctionnel

Pour cette dernière station étudiée, le programme fonctionnel a été réalisé par une personne n'ayant pas connaissance de l'existence du filtre de sécurité. Le filtre est donc un filtre correcteur « superviseur » (section I-4.3.2).

A partir du cahier des charges et d'une analyse du système, le concepteur du programme fonctionnel a structuré sa conception à l'aide de différentes tâches fonctionnelles organi-

sées de façon hiérarchique. Ces tâches fonctionnelles ont ensuite été spécifiées à l'aide de différents grafcets détaillés ci-après.

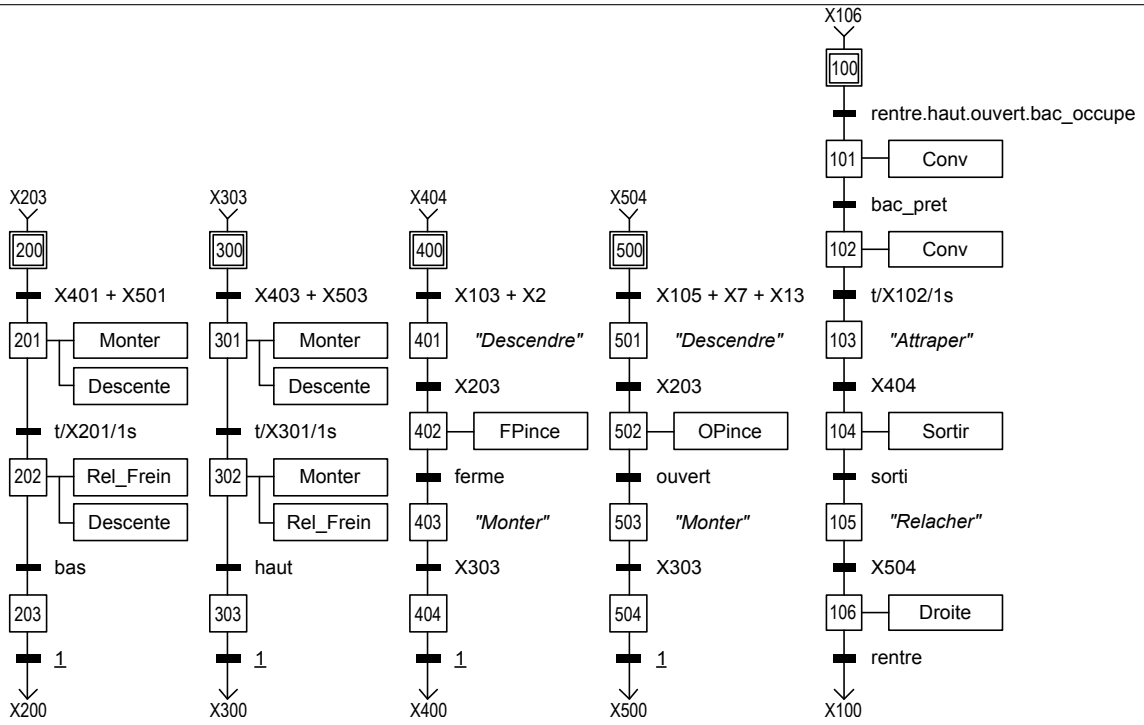
- niveau 0 :
  - Fig. 59a : Descendre le bras
  - Fig. 59b : Monter le bras
- niveau 1 (utilise le niveau 0) :
  - Fig. 59c : Attraper un 6-pack
  - Fig. 59d : Relâcher un 6-pack
- niveau 2 (utilise le niveau 1) :
  - Fig. 59e : Importer un 6-pack depuis l'entrée de la station vers le convoyeur central
  - Fig. 59f : Exporter un 6-pack depuis le convoyeur central vers une glissière de sortie

Ces grafcets ont ensuite été implémentés en langage ladder diagram (IEC61131-3, 2013) pour être intégrés dans l'automate de la station d'import/export. Le filtre de sécurité a ensuite été ajouté dans l'automate. Une IHM a également été développée afin de pouvoir activer les différentes tâches du programme fonctionnel, ainsi que pour pouvoir activer ou désactiver le filtre de sécurité.

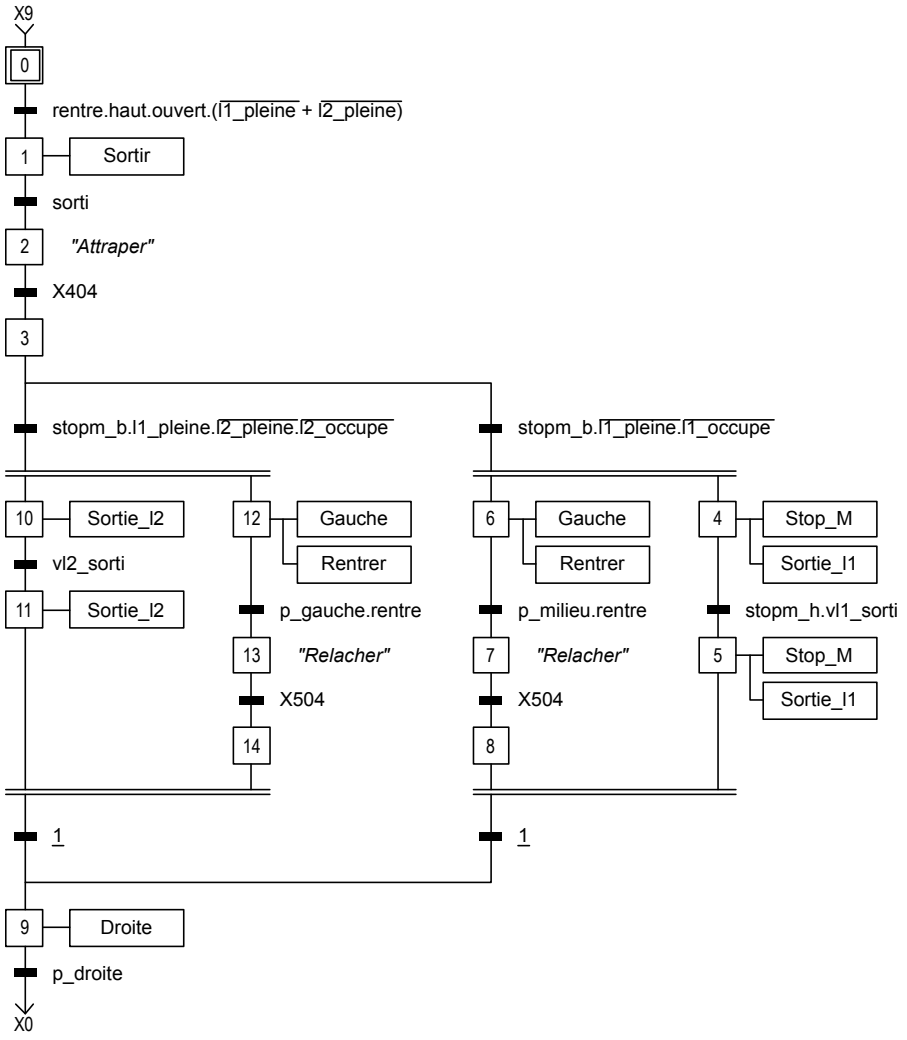
Les tests sur le système, sans le filtre de sécurité, permettent de montrer rapidement que le programme fonctionnel n'est pas entièrement bon. En effet, les grafcets réalisés ne sont pas robustes aux différents états initiaux possibles. Par exemple, la tâche « Importer » peut être exécutée alors que le système est à gauche, ce qui va engendrer un dépôt sur le convoyeur central en dehors de la zone de dépose normale.

L'activation du filtre de sécurité permet la mise en sécurité de ce programme fonctionnel présentant des erreurs. Certaines tâches s'activent et commencent leur exécution, mais certains ordres sont annulés par le filtre ce qui empêche la tâche d'atteindre un état dangereux.

Dans le cas où un programme API présente des erreurs, mais que celui-ci est trop complexe pour isoler ou corriger efficacement l'erreur, l'adjonction d'un filtre de sécurité à ce programme défectueux permet la mise en sécurité de celui-ci de façon formelle et efficace.



(a) Descendre (b) Monter (c) Attraper (d) Relacher (e) Importer



(f) Exporter

FIGURE 59 – Grafcets du programme fonctionnel pour l'import/export

V-3 Conclusion

Ce chapitre a présenté l'application de l'approche de commande par filtre logique à un système réel utilisé pour la formation et la recherche.

Dans un premier temps le système utilisé a été présenté. Ce système, appelé CellFlex, est composé de six stations différentes. Les travaux de cette thèse ont été appliqués à trois de ces stations : un convoyeur à bande, un système d'import/export et un système de mise en carton.

Par la suite, et pour chacune des stations étudiées, l'approche proposée dans cette thèse pour la conception du filtre logique a été appliquée : analyse du cahier des charges, spécifications des exigences à l'aide de contraintes logiques, analyse de la cohérence des contraintes logiques, implémentation automatique de l'algorithme du filtre logique puis intégration de celui-ci dans l'API. Le tableau 21 présente une synthèse des variables (capteurs, actionneurs, observateurs et variables internes) et des contraintes utilisées (structurelles, simples, combinées).

Table 21 – Récapitulatif

	convoyeur	transfert	import/export	total
capteurs	27	14	17	58
observateurs/mémoires	6	12	21	39
variables internes	4	3	1	8
actionneurs	9	10	13	32
contraintes simples	8	25	51	84
contraintes combinées	0	6	5	11
contraintes structurelles	0	7	12	19

Par rapport aux différents types de filtre logique présentés dans les chapitres précédents, l'approche par filtre logique correcteur (contrôleur ou superviseur) de sécurité a été retenue pour cette application. Avec cette approche, la partie fonctionnelle (ce que l'on veut faire) et la partie sécurité (ce qui est interdit) sont séparées. Le filtre logique de sécurité a donc pour objectif de garantir que tous les états dangereux ou interdits (listés dans le cahier des charges) ne puissent pas être atteints par le système, et ce quel que soit le programme fonctionnel.

Deux utilisations différentes du filtre de sécurité correcteur ont permis de montrer les

avantages de celui-ci. Ceux-ci sont résumés ci-dessous.

— Convoyeur central (filtre correcteur contrôleur) : le programme fonctionnel a été réalisé en sachant qu'un filtre de sécurité existait. La conception du programme fonctionnel est alors fortement simplifiée.

— Transfert (filtre correcteur contrôleur) : le programme fonctionnel correspond à un contrôle manuel du système au moyen d'une interface homme machine. Dans ce cas, les tests manuels sur le système peuvent s'effectuer sans risquer une détérioration du système. Dans un contexte de formation, l'apprenant gagne grandement en autonomie car la présence de l'enseignant est moins obligatoire lors des différents tests.

— Import/export (filtre correcteur superviseur) : le programme fonctionnel a été réalisé par une personne n'ayant pas connaissance du filtre de sécurité. Le programme fonctionnel résultant devait donc prendre en compte à la fois les besoins liés aux exigences fonctionnelles, mais également les interdictions liées aux exigences de sécurité. Le programme fonctionnel résultant comportait différentes erreurs, l'adjonction du filtre de sécurité à celui-ci a permis sa mise en sécurité de façon formelle et efficace.

Conclusion générale

Cette thèse contribue à une approche formelle de conception d'un programme de contrôle/commande pour les systèmes automatisés de production (SAP) contrôlés par des automates programmables industriels (API). Dans ce contexte, deux constats principaux ont été soulevés : il existe un manque de méthodologie efficace pour la conception d'un programme API dans le monde industriel et les méthodes formelles issues du monde académique ne sont ni connues ni utilisées par l'industrie car trop complexes. Par ailleurs, l'industrie du futur nécessitera des contrôleurs toujours plus flexibles et fiables. La flexibilité implique que les programmes seront encore plus difficiles à réaliser, et par conséquent, la difficulté pour garantir la fiabilité de ceux-ci sera accrue.

Pour répondre à ces problématiques, une méthode de conception formelle s'intégrant dans un cycle de développement industriel classique (cycle en V) a été proposée. De plus, afin de faciliter le transfert vers l'industrie tant d'un point de vue technique (API) que humain (pratique des automaticiens), le formalisme utilisé est entièrement basé sur des variables et des équations logiques appelées contraintes logiques. Ces contraintes logiques permettent la spécification des exigences informelles recensées dans le cahier des charges. A partir de ces contraintes logiques, un algorithme de résolution des contraintes, implémentable dans un API, est synthétisé et implémenté automatiquement dans un langage de programmation normalisé pour API. Ce filtre logique peut être utilisé pour : commander un SAP contrôlé par un API, vérifier formellement un programme API, mettre en sécurité un programme API déjà existant présentant des erreurs.

Suite aux précédents travaux sur le filtre logique (Riera *et al.*, 2015), différents verrous et points d'améliorations ont été identifiés. Les travaux de cette thèse ont eu pour objectif de lever certains verrous et de globalement améliorer et renforcer l'approche par filtre logique. Dans le but de généraliser l'approche par filtre, un effort important a été réalisé

autour de la formalisation des contraintes logiques et des différentes fonctions et propriétés associées au filtre logique. Cet apport de formalisation a permis, en particulier, de proposer une approche de vérification formelle de la notion de cohérence d'un filtre logique ainsi qu'une condition nécessaire et suffisante à cette propriété. De plus, après avoir mis à jour l'algorithme d'implémentation classique, deux algorithmes de recherche locale d'une solution basés sur des techniques de solveur SAT ont été proposés. Enfin, l'application de l'approche par filtre logique à plusieurs systèmes réels a permis de montrer ses avantages.

Améliorations possibles

Les travaux de cette thèse ont permis de généraliser, de formaliser et de globalement améliorer l'approche par filtre logique. Néanmoins, différentes voies d'améliorations sont envisageables et sont discutées dans les paragraphes suivants.

Conception des contraintes

L'application de l'approche par filtre logique à un nouveau système a pour principale difficulté la conception des contraintes logiques. En effet, le cahier des charges étant rarement complet, la formalisation des exigences peut ne pas suffire à couvrir l'ensemble du système, dès lors des contraintes supplémentaires doivent être ajoutées. De plus, lorsque plusieurs dizaines voire centaines de contraintes existent, certaines ne sont peut-être pas nécessaires. Dans le but d'aider le concepteur d'un filtre logique, différents points d'amélioration sont à envisager.

Des travaux théoriques sur l'*inclusion* de contraintes logiques doivent pouvoir être menés. En effet, une contrainte inclue dans une autre n'est pas nécessaire et peut donc être supprimée. De tels travaux devraient mener à des solutions pour réduire automatiquement un ensemble de contraintes logiques, par conséquent les analyses en seront facilitées. De cette notion d'inclusion doit pouvoir être définie une notion de *permissivité* d'une contrainte. Le but en général est d'obtenir un filtre logique le plus permissif possible, l'exemple du chat et de la souris du chapitre III illustre ce problème de permissivité. Un filtre logique le plus permissif possible permet d'obtenir un système le plus flexible possible.

Dans le but d'augmenter l'expressivité des contraintes, des variables de types différents doivent pouvoir être prises en compte. En effet, actuellement seules des exigences nécessitant des variables booléennes peuvent être traduites en contraintes logiques. A terme, l'intégration de variables entières voire réelles permettrait la prise en compte de phénomènes temporels et continus. Dès lors, les algorithmes devront être également adaptés.

Les systèmes industriels complexes sont souvent le résultat d'un assemblage d'éléments simples et normalisés (vérins, capteurs de position, convoyeurs, etc.) appelés Éléments de Parties Opératives (EPO). La modélisation de tels systèmes peut donc être effectuée en assemblant hiérarchiquement les modèles de chacun des EPO, et les modèles des interactions entre ces EPO (Philippot, 2006). Sur ce principe, il semble possible de définir et standardiser des sous-ensembles de contraintes pour chaque type d'EPO. La constitution d'un filtre logique pour un système complet reviendrait par conséquent à assembler ces sous-ensembles de contraintes. Le problème principal avec ce genre d'approche est la modélisation des différentes interactions possibles entre les EPO.

Sur ce même principe d'aide à la conception par assemblage de sous-modèles, le concept de jumeau numérique a beaucoup à apporter. Un jumeau numérique est une copie digitale d'un système, de nombreuses applications sont possibles : détection des défauts, simulation de scénarios, formation... Dans le cas de la conception d'un filtre logique, les contraintes doivent pouvoir être extraites automatiquement à partir de ce modèle numérique. En effet, en imposant des règles à respecter (issues du cahier des charges) et en simulant le système dans sa globalité, des contraintes logiques doivent pouvoir être inférées lorsque ces règles ne sont pas respectées.

Les différents algorithmes de vérification formelle de la cohérence et d'implémentation automatique du filtre logique proposés dans cette thèse ont été intégrés à l'environnement logiciel SEDMA. Néanmoins un logiciel dédié à l'approche par filtre logique permettrait de faciliter son utilisation. En effet, dans SEDMA le point d'entrée est un fichier texte décrivant les variables, les contraintes et les priorités. Dans un logiciel dédié, il serait envisageable d'avoir un éditeur dynamique de l'ensemble des contraintes avec notamment une vérification syntaxique et une coloration des variables en fonction de leurs types. De plus, un simulateur de filtre logique serait envisageable, celui-ci permettrait d'imposer une affectation initiale des variables et d'observer les cycles de résolution du filtre. Un tel

simulateur permettrait de faciliter encore plus l'utilisation de l'approche.

Analyse de la cohérence

L'approche de vérification de la cohérence nécessite actuellement une exploration complète de l'espace d'états atteignables. Dans le but de réduire le risque d'explosion combinatoire lié à cette méthode de vérification, une analyse structurelle de l'ensemble des contraintes est effectuée pour réduire l'ensemble à vérifier. Ce principe d'analyse structurelle des contraintes doit pouvoir être amélioré pour réduire encore plus l'ensemble des contraintes. Une solution peut être de coupler l'analyse structurelle actuelle avec des approches symboliques à base de diagramme de décision binaire (BDD).

Une autre solution, pour la vérification formelle de la propriété de cohérence du filtre logique, est d'utiliser l'analyse structurelle des réseaux de Petri (RdP). L'analyse structurelle des RdP semble intéressante car elle est indépendante du marquage initial. En modélisant les variables par des places et les contraintes par des transitions, il semble possible de réussir à modéliser les règles de résolution des contraintes logiques et des priorités associées. Dans ces conditions, l'analyse des T-semiflots du réseaux de Petri doit permettre de statuer sur la propriété de cohérence du filtre logique. En effet, les T-semiflots permettent de détecter des cycles potentiels de franchissement de transitions. L'analogie avec les cycles de résolution de contraintes de l'approche de vérification proposée dans cette thèse doit donc pouvoir être faite.

Implémentation dans l'API

Ces travaux de thèse ont montré qu'il est possible d'utiliser des techniques de solveur SAT pour l'implémentation d'un filtre logique dans un API. Néanmoins, les performances obtenues avec les algorithmes proposés risquent de ne pas être assez satisfaisantes pour des applications industrielles complexes. Le développement d'un algorithme de recherche locale efficace et implémentable dans un API reste donc un problème ouvert.

Perspectives d'applications

Jusqu'à maintenant, le filtre logique a été principalement appliqué à de la commande de système automatisé de production. La généralisation et la formalisation apportées avec cette thèse permettent d'envisager des applications à d'autres domaines. Par la suite, et pour clôturer cette conclusion, différentes perspectives d'applications sont discutées.

Aide à la reconfiguration

La reconfiguration d'un contrôleur consiste à changer le mode de fonctionnement d'un contrôleur suite à l'occurrence d'un événement (détection d'un défaut, changement de cahier des charges, changement de configuration du système, etc.). Un système flexible, tel que demandé par les principes de l'usine du futur, peut être vu comme un système capable de se reconfigurer efficacement. Un des problèmes de la reconfiguration est d'être capable de choisir le bon mode de fonctionnement futur du système tout en considérant le mode de fonctionnement actuel et l'état du système.

En spécifiant les règles de reconfiguration sous forme de contraintes logiques, un filtre logique de reconfiguration fournira le(s) mode(s) de fonctionnement adapté(s) à l'état actuel du système et respectant les différentes règles. Pour ce faire, les contraintes logiques de reconfiguration ne seront pas au même « niveau ». En effet, les contraintes logiques présentées dans cette thèse ont été appliquées à la commande. Par conséquent, les variables commandables correspondent aux actionneurs du système. Dans un contexte de reconfiguration, les variables commandables représenteraient les actions d'activation des modes de fonctionnement. Le filtre logique de reconfiguration permettrait donc d'empêcher ou de forcer la désactivation ou l'activation des différents modes de fonctionnement.

Les approches classiques de reconfiguration sont basées sur des modèles à états finis tels que des automates ou des réseaux de Petri (Macktoobian et Wonham, 2017). Par rapport à ces approches, différents avantages de l'approche par filtre logique, dans un contexte de reconfiguration, sont discutés ci-dessous.

— La modélisation des règles d'exclusivité entre différents modes de fonctionnement

peut être compliqué à réaliser avec les modèles à états finis. Une contrainte combinée modélise parfaitement ce type de règles.

— Le bon fonctionnement du filtre n'est que peu influencé par l'état initial du système. En effet la seule restriction est que l'état initial respecte les contraintes. Dès lors, même suite à une réinitialisation du contrôleur ou à un changement de configuration, le filtre restera opérationnel.

— L'ajout ou la suppression de règles de reconfiguration est très simple. En effet, il suffit d'ajouter ou de retirer une contrainte à l'ensemble et de générer le filtre automatiquement.

Ordonnancement et planification de tâches

Chaque type de contrainte permet de modéliser un type d'exigence. En effet, les contraintes simples modélisent aisément une exigence du type « si ... alors ... » et les contraintes combinées peuvent représenter une exclusion mutuelle. Ces types d'exigences apparaissent souvent lorsque l'on considère la planification et l'ordonnancement de tâches.

Dans un programme API, une tâche peut être considérée comme une fonction complexe dans le sens où plusieurs actions élémentaires sont nécessaires pour mener à bien la fonction, par exemple : « acheminer une pièce de la position A à la position B », « fabrication d'un mélange », « palettisation », etc.

D'un point de vue modélisation de ce type de problème, les variables commandables du filtre logique peuvent représenter les ordres d'activations d'une tâche. Les contraintes simples représenteraient les conditions d'activations d'une tâche, les relations de précédence entre tâches ou bien encore la possibilité de forcer l'activation d'une tâche après la fin d'une autre. Les contraintes combinées quand à elles permettraient de représenter facilement les exclusions mutuelles et les priorités entre différentes tâches. D'un point de vue utilisation en-ligne, le filtre logique permettrait de fournir l'ensemble des tâches autorisées, interdites ou forcées. Ce filtre d'ordonnancement pourrait grandement simplifier les problèmes de ré-ordonnancement en-ligne des programmes API.

Remontées d'informations à l'opérateur

Durant l'exploitation d'un système industriel, l'opérateur en charge de celui-ci peut être amené à observer des comportements inattendus voire dangereux. Identifier la cause de tels comportements et la localiser dans le programme peut s'avérer être très compliqué.

Lorsqu'une contrainte logique est violée, les états de capteurs et d'actionneurs violant cette contrainte peuvent fournir des informations permettant d'aider l'automaticien à corriger le programme API. Néanmoins, la spécification correcte d'une exigence nécessite assez souvent la création de plusieurs contraintes logiques. Fournir l'ensemble des états de variables violant ces contraintes ne représente donc pas une information pertinente. Dès lors, des travaux doivent être entrepris dans le but d'effectuer une remontée d'information pertinente à l'utilisateur lorsque le système possède un filtre logique.

Bibliographie

ADEPA : Le GEMMA, Guide d'Étude des Modes de Marches et d'Arrêts, 1981. URL https://www.technologuepro.com/cours-automate-programmable-industriel/GEMMA/Gemma-original.pdf.

K. Akesson, M. Fabian, H. Flordal et R. Malik : Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. *In 8th International Workshop on Discrete Event Systems (WODES'06)*, pages 384–385, juillet 2006.

A. Bajovs, O. Nikiforova et J. Sejans : Code Generation from UML Model : State of the Art and Practical Implications. *Applied Computer Systems*, 14(1):9–18, juin 2013. URL https://content.sciendo.com/view/journals/acss/14/1/article-p9.xml.

S. Balemi, G.J. Hoffmann, P. Gyugyi, H. Wong-Toi et G.F. Franklin : Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, juillet 1993. ISSN 0018-9286.

T. Balyo, M. Heule et M. Jarvisalo : SAT Competition 2017 : Solver and Benchmark Descriptions, 2017. URL https://helda.helsinki.fi/bitstream/handle/10138/224324/sc2017-proceedings.pdf?sequence=4.

P. Baptiste, C. Le Pape et W. Nuijten : *Constraint Based Scheduling : Applying Constraint Programming to Scheduling Problems*. Springer Science & Business Media, décembre 2012. ISBN 978-1-4615-1479-4. Google-Books-ID : qUzhBwAAQBAJ.

K. Beck et E. Gamma : *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, 2000. ISBN 978-0-201-61641-5. Google-Books-ID : G8EL4H4vf7UC.

R. Benlorhfar, D. Annebicque, F. Gellot et B. Riera : Robust filtering of PLC program for automated systems of production. *In 18th World Congress of the International Federation of Automatic Control*, 2011.

B. Berthomieu, P.-O. Ribet et F. Vernadat : The tool TINA – Construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, juillet 2004. ISSN 0020-7543. URL http://dx.doi.org/10.1080/00207540412331312688.

B. W. Boehm : Improving Software Productivity. *Computer*, pages 43–47, 1987.

B. W. Boehm : A spiral model of software development and enhancement. *Computer*, 21(5):61–72, mai 1988. ISSN 0018-9162.

G. Boole : *An Investigation of the Laws of Thought : On which are Founded the Mathematical Theories of Logic and Probabilities*. Dover Publications, 1854. Google-Books-ID : pn_HORLo1uIC.

J. P. Bowen et M. G. Hinchey : Ten Commandments Revisited : A Ten-year Perspective on the Industrial Application of Formal Methods. *In Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 8–16, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-148-1. URL http://doi.acm.org/10.1145/1081180.1081183.

B. A. Brandin et W. M. Wonham : Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, février 1994. ISSN 0018-9286.

K. Cai et W. M. Wonham : Supervisor Localization : A Top-Down Approach to Distributed Control of Discrete-Event Systems. *IEEE Transactions on Automatic Control*, 55(3):605–618, mars 2010. ISSN 0018-9286.

M. Cantarelli et J. M. Roussel : Reactive control system design using the Supervisory Control Theory : Evaluation of possibilities and limits. *In 2008 9th International Workshop on Discrete Event Systems*, pages 200–205, mai 2008.

C. G. Cassandras et S. Lafortune : *Introduction to Discrete Event Systems*. Springer Science & Business Media, décembre 2009. ISBN 978-0-387-33332-8.

F. Charbonnier, H. Alla et R. David : Discrete-event dynamic systems. *Control Systems Technology, IEEE Transactions on*, 7(2):175–187, 1999.

M. Combacau : Support de cours sur les réseaux de Petri, 2013. URL http://homepages.laas.fr/combacau/enseignements/SEDMA-RdP-M1ISTR-2014.pdf.

M. Combacau, P. Esteban et A. Nketsa : Commandes à réseaux de Petri - Modélisation, mars 2005. URL https://www.techniques-ingenieur.fr/base-documentaire/automatique-robotique-th16/automatique-sequentielle-42395210/commandes-a-reseaux-de-petri-s7572/.

R. Coupat : *Méthodologie pour les études d'automatisation et la génération automatique de programmes Automates Programmables Industriels sûrs de fonctionnement. Application aux Equipements d'Alimentation des Lignes Électrifiées*. Reims, novembre 2014. URL http://www.theses.fr/2014REIMS019.

R. Coupat, A. Philippot, M. Niang, C. Courtois, D. Annebicque et B. Riera : Methodology for Railway Automation Study and Automatic Generation of PLC Programs. *IEEE Intelligent Transportation Systems Magazine*, pages 1–1, 2018. ISSN 1939-1390.

D. Cruette, J. P. Bourey et J. C. Gentina : Hierarchical specification and validation of operating sequences in the context of FMSs. *Computer Integrated Manufacturing Systems*, 4(3):140–156, août 1991. ISSN 0951-5240. URL http://www.sciencedirect.com/science/article/pii/095152409190043X.

R. Cuer, L. Piétrac, E. Niel, S. Diallo, N. Minoiu-Enache et C. Dang-Van-Nhan : A formal framework for the safe design of the Autonomous Driving supervision. *Reliability Engineering & System Safety*, 174:29–40, juin 2018. ISSN 0951-8320. URL http://www.sciencedirect.com/science/article/pii/S0951832017305914.

E. Dallal, A. Colombo, D. D. Vecchio et S. Lafortune : Supervisory control for collision avoidance in vehicular networks using discrete event abstractions. *Discrete Event Dynamic Systems*, 27(1):1–44, mars 2017. ISSN 0924-6703, 1573-7594. URL https://link.springer.com/article/10.1007/s10626-016-0228-3.

R. David : Grafcet : a powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, 3(3):253–268, 1995. ISSN 1063-6536.

M. Davis, G. Logemann et D. Loveland : A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, juillet 1962. ISSN 0001-0782. URL http://doi.acm.org/10.1145/368273.368557.

D. Du, J. Gu et P. M. Pardalos : *Satisfiability Problem : Theory and Applications : DIMACS Workshop, March 11-13, 1996*. American Mathematical Soc., janvier 1997. ISBN 978-0-8218-7080-8.

J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North et G. Woodhull : Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. *In Graph Drawing Software*, Mathematics and Visualization, pages 127–148. Springer, Berlin, Heidelberg, 2004. ISBN 978-3-642-62214-4 978-3-642-18638-7. URL https://link.springer.com/chapter/10.1007/978-3-642-18638-7_6.

N. Eén et N. Sörensson : An Extensible SAT-solver. *In Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 502–518. Springer, Berlin, Heidelberg, mai 2003. ISBN 978-3-540-20851-8 978-3-540-24605-3. URL https://link.springer.com/chapter/10.1007/978-3-540-24605-3_37.

M. Fabian et A. Hellgren : PLC-based implementation of supervisory control for discrete event systems. *In Proceedings of the 37th IEEE Conference on Decision and Control, 1998*, volume 3, pages 3305–3310 vol.3, 1998.

G. Faraut, L. Pietrac et E. Niel : Control law synthesis and reconfiguration using SCT. *In 2010 Conference on Control and Fault-Tolerant Systems (SysTol)*, pages 576–581, octobre 2010.

Z. Fei, S. Miremadi, K. Åkesson et B. Lennartson : Efficient Symbolic Supervisor Synthesis for Extended Finite Automata. *IEEE Transactions on Control Systems Technology*, 22(6):2368–2375, novembre 2014. ISSN 1063-6536.

H. Flordal, R. Malik, M. Fabian et K. Åkesson : Compositional Synthesis of Maximally Permissive Supervisors Using Supervision Equivalence. *Discrete Event Dynamic Systems*, 17(4):475–504, novembre 2007. ISSN 0924-6703, 1573-7594. URL http://link.springer.com/10.1007/s10626-007-0018-z.

D. Foures : *Validation de modèles de simulation*. Thèse de doctorat, Toulouse 3, juin 2015. URL http://www.theses.fr/2015TOU30071.

G. Frey et L. Litz : Formal methods in PLC programming. *In 2000 IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 2431–2436 vol.4, 2000.

J. Girault, J.-J. Loiseau et O. H. Roux : Online compositional controller synthesis for AGV. *Discrete Event Dynamic Systems*, 26(4):583–610, décembre 2016. ISSN 0924-6703, 1573-7594. URL https://link.springer.com/article/10.1007/s10626-015-0222-1.

P. Gohari et W. M. Wonham : On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 30(5):643–652, octobre 2000. ISSN 1083-4419.

D. Gouyon, J.-F. Pétin et A. Gouin : A pragmatic approach for modular control synthesis and implementation. *International Journal of Production Research*, 2(14): 2839–2858, 2004. URL https://hal.archives-ouvertes.fr/hal-00120780.

D. Gouyon, J.-F. Pétin et G. Morel : A product-driven reconfigurable control for shop floor systems. *Studies in Informatics and Control*, 16(1):??–??, mars 2007. URL https://hal.archives-ouvertes.fr/hal-00134745/document.

A. Grastien et A. Anbulagan : Diagnosis of Discrete Event Systems Using Satisfiability Algorithms : A Theoretical and Empirical Study. *IEEE Transactions on Automatic Control*, 58(12):3070–3083, décembre 2013. ISSN 0018-9286.

J. Gu, P. W. Purdom, J. Franco et B. W. Wah : Algorithms for the Satisfiability (SAT) Problem. *In* D.-Z. Du et P. M. Pardalos, éditeurs : *Handbook of Combinatorial Optimization*, pages 379–572. Springer US, 1999. ISBN 978-1-4419-4813-7 978-1-4757-3023-4. URL http://link.springer.com/chapter/10.1007/978-1-4757-3023-4_7.

H. M. Hanisch, J. Thieme, A. Luder et O. Wienhold : Modeling of PLC behavior by means of timed net condition/event systems. *In , 1997 6th International Conference on Emerging Technologies and Factory Automation Proceedings, 1997. ETFA '97*, pages 391–396, septembre 1997.

R. M. Haralick et G. L. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, octobre 1980. ISSN 0004-3702. URL http://www.sciencedirect.com/science/article/pii/000437028090051X.

Y. Hietter : *Synthèse algébrique de lois de commande pour les systèmes à évènements discrets logiques*. Thèse de doctorat, École normale supérieure de Cachan-ENS Cachan, 2009.

J. R. Highsmith : *Adaptive Software Development : A Collaborative Approach to Managing Complex Systems*. Addison-Wesley, juillet 2013. ISBN 978-0-13-348948-4. Google-Books-ID : CVcUAAAAQBAJ.

R. C. Hill, J. E. R. Cury, M. H. de Queiroz, D. M. Tilbury et S. Lafortune : Multi-level hierarchical interface-based supervisory control. *Automatica*, 46(7):1152–1164, juillet 2010. ISSN 0005-1098. URL http://www.sciencedirect.com/science/article/pii/S0005109810001597.

IEC61131 : Programmable controllers - ALL PARTS, 2018. URL https://webstore.iec.ch/publication/62427.

IEC61131-3 : Programming languages for programmable logic controllers, 2013.

ISO15408 : Information technology – Security techniques – Evaluation criteria for IT security – Part 1 : Introduction and general model, 2009. URL https://www.iso.org/standard/50341.html.

ISO9000 : Systèmes de management de la qualité - Principes essentiels et vocabulaire., 2015. URL https://www.iso.org/obp/ui/fr/#iso:std:iso:9000:ed-4:v1:en.

K. Jensen, L. M. Kristensen et L. Wells : Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, juin 2007. ISSN 1433-2779, 1433-2787. URL https://link.springer.com/article/10.1007/s10009-007-0038-x.

T. L. Johnson : Improving automation software dependability : A role for formal methods ? *Control Engineering Practice*, 15(11):1403–1415, novembre 2007. ISSN 0967-0661. URL http://www.sciencedirect.com/science/article/pii/S0967066106001249.

N. Jussien, G. Rochart et X. Lorca : Choco : an Open Source Java Constraint Programming Library. *In CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008. URL https://hal.archives-ouvertes.fr/hal-00483090/document.

C. G. Lee et S. C. Park : Survey on the virtual commissioning of manufacturing systems. *Journal of Computational Design and Engineering*, 1(3):213–222, juillet 2014. ISSN 2288-4300. URL http://www.sciencedirect.com/science/article/pii/S2288430014500292.

M. Lhommeau : *Etude de systèmes à événements discrets dans l'algèbre (max,+) : synthèse de correcteurs robustes dans un dioïde d'intervalles : synthèse de correcteurs en présence de perturbations*. Thèse de doctorat, Angers, janvier 2003. URL http://www.theses.fr/2003ANGE0017.

D. Lime, O. H. Roux, C. Seidner et L.-M. Traonouez : Romeo : A Parametric Model-Checker for Petri Nets with Stopwatches. *In Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 54–57. Springer, Berlin, Heidelberg, mars 2009. ISBN 978-3-642-00767-5 978-3-642-00768-2. URL https://link.springer.com/chapter/10.1007/978-3-642-00768-2_6.

M. Macktoobian et W. M. Wonham : Automatic reconfiguration of untimed discrete-event systems. *In Electrical Engineering, Computing Science and Automatic Control (CCE)*, pages 1–6. IEEE, octobre 2017. ISBN 978-1-5386-3406-6. URL http://ieeexplore.ieee.org/document/8108839/.

A. K. Mackworth : Consistency in Networks of Relations. *In* B. L. Webber et N. J. Nilsson, éditeurs : *Readings in Artificial Intelligence*, pages 69–78. Morgan Kaufmann, janvier 1981. ISBN 978-0-934613-03-3. URL http://www.sciencedirect.com/science/article/pii/B978093461303350009X.

A. Mader : A Classification of PLC Models and Applications. *In Discrete Event Systems*, The Springer International Series in Engineering and Computer Science, pages 239–246. Springer, Boston, MA, 2000. ISBN 978-1-4613-7025-3 978-1-4615-4493-7. URL https://link.springer.com/chapter/10.1007/978-1-4615-4493-7_24.

P. Magnusson, M. Fabian et K. Åkesson : Modular specification of forbidden states for supervisory control. *IFAC Proceedings Volumes*, 43(12):412–417, janvier 2010. ISSN 1474-6670. URL http://www.sciencedirect.com/science/article/pii/S1474667015324903.

P. Marangé : *Synthesis and robust filtering of control for dependable manufacturing systems*. Theses, Université de Reims - Champagne Ardenne, novembre 2008. URL https://tel.archives-ouvertes.fr/tel-00353654.

P. Marangé, F. Gellot et B. Riera : Remote Control of Automation Systems for DES Courses. *IEEE Transactions on Industrial Electronics*, 54(6):3103–3111, décembre 2007. ISSN 0278-0046.

T. J. McCabe : A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, décembre 1976. ISSN 0098-5589.

G. H. Mealy : A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. ISSN 1538-7305. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1955.tb03788.x.

M. G. Mehrabi, A. G. Ulsoy et Y. Koren : Reconfigurable manufacturing systems : Key to future manufacturing. *Journal of Intelligent Manufacturing*, 11(4):403–419, août 2000. ISSN 0956-5515, 1572-8145. URL https://link.springer.com/article/10.1023/A:1008930403506.

S. Miremadi, K. Akesson, M. Fabian, A. Vahidi et B. Lennartson : Solving two supervisory control benchmark problems using Supremica. *In Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 131–136. IEEE, 2008. URL http://ieeexplore.ieee.org/abstract/document/4605934/.

S. Miremadi, K. Akesson et B. Lennartson : Symbolic Computation of Reduced Guards in Supervisory Control. *IEEE Transactions on Automation Science and Engineering*, 8(4):754–765, 2011. ISSN 1545-5955.

S. Miremadi et A. Voronov : Symbolic reduction of guards in supervisory control using genetic algorithms, 2012. URL http://publications.lib.chalmers.se/records/fulltext/local_164304.pdf.

S. Mohajerani, R. Malik et M. Fabian : A Framework for Compositional Synthesis of Modular Nonblocking Supervisors. *IEEE Transactions on Automatic Control*, 59 (1):150–162, janvier 2014. ISSN 0018-9286.

E. F. Moore : Machine models of self-reproduction. *Proceedings of symposia in applied mathematics*, 14(1962):17–33, 1962.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang et S. Malik : Chaff - Engineering an Efficient SAT Solver. *In Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 978-1-58113-297-7. URL http://doi.acm.org/10.1145/378239.379017.

M. Niang, A. Philippot, F. Gellot, R. Coupat, B. Riera et S. Lefebvre : Formal Verification for Validation of PSEEL's PLC Program :. *In 14th International Conference on Informatics in Control, Automation and Robotics*, pages 567–574. SCITEPRESS - Science and Technology Publications, 2017. ISBN 978-989-758-263-9 978-989-758-264-6. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006418705670574.

M. Pellicciari, A. O. Andrisano, F. Leali et A. Vergnano : Engineering method for adaptive manufacturing systems design. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 3(2):81–91, mai 2009. ISSN 1955-2513, 1955-2505. URL https://link.springer.com/article/10.1007/s12008-009-0065-9.

Y. Pencolé, R. Pichard et P. Fernbach : Modular fault diagnosis in discrete-event systems with a CPN diagnoser. *In 9th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes*, Paris, 2015.

C. A. Petri : Kommunikation mit Automaten. *http ://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/pdf/diss_petri_d.pdf*, 1962. URL http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

A. Philippot : *Contribution au diagnostic décentralisé des systèmes à événements discrets : Application aux systèmes manufacturiers*. PhD Thesis, Université de Reims-Champagne Ardenne, 2006.

R. Pichard, N. Ben Rabah, V. Carre-Menetrier et B. Riera : CSP solver for Safe PLC Controller : Application to manufacturing systems. *In IFAC MIM'16*, volume 49 de *8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016Troyes, France, 28—30 June 2016*, pages 402–407, Troyes, 2016. URL http://www.sciencedirect.com/science/article/pii/S2405896316309119.

R. Pichard, M. Combacau, A. Philippot, R. Saddem et B. Riera : SEDMA - un outil pour la Modélisation, l'Analyse et la génération automatique de programme pour les SEDs. *In MSR 2017, 11ème Colloque sur la Modélisation des Systèmes Réactifs*, page 2, Marseille, novembre 2017a.

R. Pichard, A. Philippot et B. Riera : Consistency Checking of Safety Constraints for Manufacturing Systems with Graph Analysis. *In 20th IFAC WorldCongress*, volume 50 de *20th IFAC World Congress*, pages 1193–1198, Toulouse, juillet 2017b. URL http://www.sciencedirect.com/science/article/pii/S2405896317305906.

R. Pichard, A. Philippot et B. Riera : Safe PLC Controller implementation IEC 61131-3 compliant based on a simple SAT solver : Application to manufacturing systems. *In ICINCO*, Porto, juillet 2018a.

R. Pichard, A. Philippot, R. Saddem et B. Riera : Safety of Manufacturing Systems Controllers by Logical Constraints With Safety Filter. *IEEE Transactions on Control Systems Technology*, pages 1–9, 2018b. ISSN 1063-6536.

L. Piétrac : Méthodes pour la conception sûre de contrôleurs discrets : apports théoriques et applications (HDR), 2018.

P. Ponsa, R. Vilanova et B. Amante : Towards integral human-machine system conception : From automation design to usability concerns. *In 2009 2nd Conference on Human System Interactions*, pages 427–433, mai 2009.

P.J.G. Ramadge et W.M. Wonham : The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, janvier 1989. ISSN 0018-9219.

L. Ricker, S. Lafortune et S. Genc : DESUMA : A Tool Integrating GIDDES and UMDES. *In 8th International Workshop on Discrete Event Systems*, pages 392–393, 2006.

B. Riera, F. Emprin, D. Annebicque, M. Colas et B. Vigário : HOME I/O : a virtual house for control and STEM education from middle schools to Universities. *IFAC-PapersOnLine*, 49(6):168–173, 2016. ISSN 2405-8963. URL `https://www.sciencedirect.com/science/article/pii/S2405896316303779`.

B. Riera, A. Philippot, D. Annebicque et F. Gellot : La commande par contraintes logiques de sécurité : principe, applications et mise en oeuvre. *In Modélisation des Systèmes Réactifs (MSR 2015)*, novembre 2015. URL `https://hal.inria.fr/hal-01224261/document`.

M. Roth, L. Litz et J.-J. Lesage : Identification of Discrete Event Systems : Implementation Issues and Model Completeness. *In 7th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 3, pages 73–80, Funchal, Portugal, juin 2010. URL `https://hal.archives-ouvertes.fr/hal-00525602`.

J.-M. Roussel et A. Giua : DESIGNING DEPENDABLE LOGIC CONTROLLERS USING THE SUPERVISORY CONTROL THEORY. *IFAC Proceedings Volumes*, 38 (1):56–61, 2005. ISSN 14746670. URL `http://linkinghub.elsevier.com/retrieve/pii/S1474667016374468`.

J.-M. Roussel et J.-J. Lesage : Design of Logic Controllers Thanks to Symbolic Computation of Simultaneously Asserted Boolean Equations. *Mathematical Problems in Engineering*, 2014:Article ID 726246, mai 2014. 15 pages.

W. W. Royce : Managing the development of large software systems : concepts and techniques. *In Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen et D. C. Teneketzis : Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, mars 1996. ISSN 1063-6536.

K. Schwaber : *Agile Project Management with Scrum*. Microsoft Press, février 2004. ISBN 978-0-7356-3790-0. Google-Books-ID : 6pZCAwAAQBAJ.

K. Schwaber et K. Mar : *Agile software development with Scrum*. Series in agile software development. Prentice Hall, Upper Saddle River, NJ, 2002. ISBN 978-0-13-067634-4.

B. Selman, H. Kautz et B. Cohen : Local Search Strategies for Satis ability Testing. *Cliques, coloring, and satisfiability*, 26(1):521–532, 1995.

K. T. Seow : Organizational Control of Discrete-Event Systems : A Hierarchical Multi-world Supervisor Design. *IEEE Transactions on Control Systems Technology*, 22(1):23–33, janvier 2014. ISSN 1063-6536.

F. Sicard, J.-M. Flaus et E. Zamaï : Approche filtre basée sur la notion de distance pour la détection des cyberattaques. *In 15ème Colloque national AIP-Primeca*, pages 1–5, La Plagne, France, avril 2017. AIP-Priméca. URL `https://hal.archives-ouvertes.fr/hal-01562589`.

P. Taillard : L'analyse structurée par tâches. *Technologie*, 170:6, 2012. URL `https://www.ac-paris.fr/portail/upload/docs/application/pdf/2012-01/articletechnologie170_analyse_par_taches.pdf`.

A. Theorin : *A Sequential Control Language for Industrial Automation*, volume TFRT-1104. Department of Automatic Control, Lund Institute of Technology, Lund University, 2014. ISBN 978-91-7623-111-1 978-91-7623-110-4. URL `http://lup.lub.lu.se/record/4698754`.

M. Tiller : *Introduction to Physical Modeling with Modelica*. Springer Science & Business Media, décembre 2012. ISBN 978-1-4615-1561-6. Google-Books-ID : 7vcGCAAAQBAJ.

T. Toublanc, S. Guillet, F. de Lamotte, P. Berruet et V. Lapotre : Using a Virtual Plant to Support the Development of Intelligent Gateway for Sensors/Actuators Security. *IFAC-PapersOnLine*, 50(1):5837–5842, juillet 2017. ISSN 2405-8963. URL `http://www.sciencedirect.com/science/article/pii/S2405896317309096`.

A. Vahidi, M. Fabian et B. Lennartson : Efficient supervisory synthesis of large systems. *Control Engineering Practice*, 14(10):1157–1167, octobre 2006. ISSN 09670661. URL `http://linkinghub.elsevier.com/retrieve/pii/S0967066106000207`.

A. D. Vieira, E. A. P. Santos, M. H. de Queiroz, A. B. Leal, A. D. de Paula Neto et J. E. R. Cury : A Method for PLC Implementation of Supervisory Control of Discrete Event Systems. *IEEE Transactions on Control Systems Technology*, 25(1):175–191, janvier 2017. ISSN 1063-6536.

A.D. Vieira, J.E.R. Cury et M.H. de Queiroz : A Model for PLC Implementation of Supervisory Control of Discrete Event Systems. *In IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06*, pages 225–232, septembre 2006.

J. N. Vilela et P. N. Pena : Supervisor abstraction to deal with planning problems in manufacturing systems. *In 2016 13th International Workshop on Discrete Event Systems (WODES)*, pages 117–122, 2016.

A. Voronov et K. Akesson : Supervisory control using satisfiability solvers. *In 2008 9th International Workshop on Discrete Event Systems*, pages 81–86, mai 2008.

R. Wang, X. Song, J. Zhu et M. Gu : Formal modeling and synthesis of programmable logic controllers. *Computers in Industry*, 62(1):23–31, janvier 2011. ISSN 0166-3615. URL `http://www.sciencedirect.com/science/article/pii/S0166361510000886`.

K. C. Wong et W. M. Wonham : Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems*, 6(3):241–273, juillet 1996. ISSN 0924-6703, 1573-7594. URL https://link.springer.com/article/10.1007/BF01797154.

W. M. Wonham : Supervisory Control of Discrete-Event Systems. *In Encyclopedia of Systems and Control*, pages 1396–1404. Springer, London, 2015. URL https://link.springer.com/referenceworkentry/10.1007/978-1-4471-5058-9_54.

T.-S. Yoo et S. Lafortune : A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems. *Discrete Event Dynamic Systems*, 12(3):335–377, juillet 2002. ISSN 0924-6703, 1573-7594. URL https://link.springer.com/article/10.1023/A:1015625600613.

M. Zanella et G. Lamperti : Continuous diagnosis of discrete-event systems. *In proceedings of the 14th International Workshop on Principles of Diagnosis (DX-03), Washington, DC, USA*, pages 105–111, 2003. URL https://pdfs.semanticscholar.org/80aa/8257bcb631f09602c5a440d8b8d61ef74d2d.pdf.

J. Zaytoon : *Discrete Event Systems 2004 (WODES'04) : A Proceedings Volume from the 7th IFAC Workshop, Reims, France, 22-24 September 2004*. Elsevier, 2005. ISBN 978-0-08-044168-9. Google-Books-ID : yOK2i7XmpUoC.

J. Zaytoon et B. Riera : Synthesis and implementation of logic controllers – A review. *Annual Reviews in Control*, mars 2017. ISSN 13675788. URL http://linkinghub.elsevier.com/retrieve/pii/S1367578816301043.

J. Zhang, M. Khalgui, Z. Li, G. Frey, O. Mosbahi et H. B. Salah : Reconfigurable Coordination of Distributed Discrete Event Control Systems. *IEEE Transactions on Control Systems Technology*, 23(1):323–330, janvier 2015. ISSN 1063-6536.

ANNEXE A

Rappels et définitions

I-1 Les automates à états finis

Les automates à états décrivent les SED comme un ensemble d'événements Σ associé à un ensemble d'états X. Ces événements font évoluer le système d'un état à un autre. Les SED sont basés sur l'alphabet d'un langage où les séquences d'événements sont des mots de ce langage. On appelle « automate » un dispositif qui engendre un langage en manipulant l'alphabet (les événements).

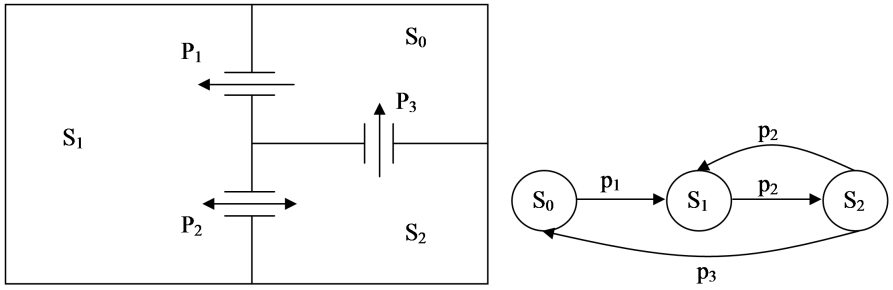
Définition 36. ***Automate à états finis déterministe***
Un automate à états finis déterministe est un 5-uplet $G = \langle X, \Sigma, \delta, x_0, X_m \rangle$, tel que :

- *X est l'ensemble fini des états qui constitue l'espace discret des états ;*
- *Σ est un alphabet fini décrivant l'ensemble des événements ;*
- *δ est la fonction de transition d'état : $\delta : X \times \Sigma \to X$;*
- *x_0 est l'état initial unique ;*
- *X_m est l'ensemble des états marqués qui définissent les états finaux, $X_m \subseteq X$.*

La figure 60 présente un système où une souris se déplace de manière spontanée à l'intérieur d'une maison (Fig. 60a). Les salles S_i, où $i \in \{0, 1, 2\}$, communiquent par des portes unidirectionnelles P_1 et P_3 et bidirectionnelle P_2 et p_i définit l'événement : « la souris passe par la porte P_i ».

Soit Σ l'ensemble des événements : $\Sigma = \{p_1, p_2, p_3\}$. Les situations possibles correspondent à la présence de la souris dans la salle S_0, S_1 ou S_2. Ceci défini trois états différents, relatifs aux possibilités d'occupation des salles. Le passage de l'état S_0 « souris en S_0 »à l'état S_1 « souris en S_1 » a lieu suite à l'occurrence de l'événement p_1 « la souris passe par la porte P_1 ». Le modèle discret du système peut être construit (Fig. 60b) et l'espace d'états s'écrit alors $X = \{S_0, S_1, S_2\}$.

Les automates à états finis sont basés sur des notions de langages et d'événements permettant la manipulation d'algorithmes mathématiques. Par la suite, quelques rappels sur la formalisation des langages et des automates à états sont présentés.



(a) La maison (b) Modèle automate du système

FIGURE 60 – Exemple de la souris dans une maison

I-2 Notion de Langages

Un langage L est un ensemble de mots, ou de chaînes, formés à partir des événements définis sur un alphabet Σ. La longueur d'un mot s, notée $|s|$, correspond au nombre d'événements qui le composent. Soit $\Sigma = \{\alpha, \beta, \gamma\}$ un alphabet, un langage de cet ensemble peut être $L1 = \{\epsilon, \alpha, \alpha\beta\beta\}$ constitué de 3 mots où le mot vide est noté ϵ, soit $|\epsilon| = \emptyset$.

Un langage L engendré par un automate à états fini G est dit régulier s'il inclut toutes les traces qui peuvent être exécutées à partir de l'état initial de G. Il est noté $L(G)$ et est défini par :

$$L(G) = \{s \in \Sigma^* | \delta(x_0, s) \neq \emptyset\}$$

où Σ^* est l'ensemble des séquences d'événements de longueur finie.

Une chaîne s_1 est dite préfixe d'une chaîne s_3 s'il existe une chaîne $s_2 \in \Sigma^*$ telle que $s_1 s_2 = s_3$. Dès lors, une chaîne est préfixe d'elle-même. Un langage $L \subseteq \Sigma^*$ est dit préfixe clos, et noté $pr(L)$, s'il comprend tous les préfixes des chaînes de L.

$$pr(L) = \{s_1 \in \Sigma^* | \exists s_2 \in \Sigma^*, s_1 s_2 \in L\}$$

Un langage L est fermé par ses préfixes (ou préfixe clos) si $L = pr(L)$. Par exemple, si $\Sigma = \{\alpha, \beta\}$ et pour $L = \{\epsilon, \alpha, \alpha\beta, \alpha\beta\beta\}$, la chaîne $\alpha\beta$ est un préfixe de la chaîne $\alpha\beta\beta$ alors $L = pr(L)$. Par contre, $L = \{\epsilon, \alpha, \alpha\beta\beta\}$ n'est pas préfixe clos, d'où $L \neq pr(L)$.

L'ensemble des traces bloquantes de L est défini comme les traces n'ayant plus d'extensions dans L, c'est-à-dire, $s \in L$ est une trace bloquante si $\{s\}\Sigma^* \cup L = \{s\}$. Un état atteint par une trace bloquante est alors appelé état bloquant.

Le langage marqué $L_m(G)$ constitue tous les mots de l'automate G dont l'exécution traduit la réalisation d'une tâche. En effet, pour un automate fini $G = (X, \Sigma, \delta, x_0, X_m)$, l'ensemble des chaînes $s \in \Sigma^*$, telles que $\delta(x_0, s) \in X$, définissent le langage généré par G et est noté $L(G)$. Si maintenant $s \in L(G)$ tel que $\delta(x_0, s) \in X_m$, alors le langage est dit marqué par G et est noté $L_m(G)$. Par conséquent, $L_m(G) \subseteq L(G)$ pour $L(G) \subseteq \Sigma^*$ et pour $L(G)$ non vide. Dès lors, $L(G)$ est dit préfixe clos et contient $L_m(G)$ défini par :

$$L_m(G) = \{s \in L(G) : \delta(x_0, s) \in X_m\}$$

A partir de la notion d'état bloquant, et du langage marqué, il est possible de définir qu'un automate est non-bloquant s'il est toujours possible d'atteindre un état marqué (Déf. 37).

Définition 37. ***Automate non-bloquant***
Un automate $A = \langle X, \Sigma, \delta, x_0, X_m \rangle$ est non-bloquant si, pour chaque trace s et chaque état $x \in X$ tel que $x_0 \xrightarrow{s} x$, il existe une trace t telle que $x \xrightarrow{t} x_m \in X_m$.

Par conséquent, un automate peut également être qualifié de deadlock lorsque celui-ci peut se retrouver indéfiniment bloqué dans un état (marqué ou non) (Déf. 38).

Définition 38. ***Automate non-deadlock***
Un automate $A = \langle X, \Sigma, \delta, x_0, X_m \rangle$ est non-deadlock si, pour chaque trace s et chaque état $x \in X$ tel que $x_0 \xrightarrow{s} x$, il existe une trace t telle que $x \xrightarrow{t} q \in X$ avec $q \neq x$.

ANNEXE B

SEDMA - un outil logiciel pour la Modélisation et l'Analyse des SEDs

II-1 Introduction

Les Systèmes à Événements Discrets (SED) permettent la modélisation (Cassandras et Lafortune, 2009) et le contrôle (Zaytoon et Riera, 2017) d'une large gamme de systèmes. Il existe différents outils logiciels permettant la modélisation et l'analyse des SEDs : TINA (Berthomieu *et al.*, 2004), Romeo (Lime *et al.*, 2009), Supremica (Akesson *et al.*, 2006), DESUMA (Ricker *et al.*, 2006), CPNtools (Jensen *et al.*, 2007), Grafchart (Theorin, 2014), etc. Ces outils sont très performants et en général spécialisés sur un seul formalisme. Néanmoins pour l'enseignement des SEDs il est souvent nécessaire de travailler sur différents formalismes, la prise en main d'un logiciel spécialisé peut être compliqué et par conséquent rajouter une difficulté de compréhension à l'apprenant. Enfin, la plupart du temps ces logiciels ne permettent pas l'implémentation des modèles créés vers un langage de programmation, il est donc parfois difficile de passer de la théorie à la pratique.

SEDMA (Pichard *et al.*, 2017a) a pour but de proposer un environnement unique pour l'apprentissage de plusieurs formalismes autour des SED. Actuellement il est possible de travailler sur les automates à états finis, les réseaux de Petri interprétés et les contraintes logiques (Pichard *et al.*, 2018b). Pour chacun de ces formalismes différentes fonctionnalités d'analyse, de conception et de simulation sont disponibles.

Néanmoins SEDMA n'a pas pour ambition de proposer l'ensemble des fonctionnalités présentes dans les logiciels spécialisés, c'est pourquoi il est possible d'échanger les modèles avec ces logiciels via un mécanisme d'import/export.

Afin de faciliter le passage à la pratique, SEDMA propose une transformation automatique des modèles vers différents langages de programmation. Actuellement il est possible de générer du code C, ST et VHDL.

Il est à noter que historiquement, SEDMA a été développé par Pierre Fernbach et Romain Pichard dans le cadre d'un projet de Master 1 à l'Université Paul Sabatier de Toulouse encadré par le Professeur Michel Combacau. Puis, dans le cadre de cette thèse, une restructuration a été apportée au logiciel par l'encadrement de 2 stages de Master 1 à l'Université de Reims Champagne-Ardenne. Cette restructuration été nécessaire afin de

pouvoir plus facilement le faire évoluer et y intégrer les travaux développés dans cette thèse (contraintes logiques).

Le but de cette annexe est de présenter les fonctionnalités de SEDMA. Pour ce faire, l'interface graphique est présentée. Puis, les fonctionnalités principales pour chacun des formalismes disponibles sont présentées.

II-2 L'interface graphique

L'interface utilisateur (Fig. 61) se veut simple et intuitive. Elle propose un ensemble de menus et d'outils, une vue arborescente sous forme de projet contenant les différents fichiers le composant (modèles graphiques, fichiers textes, images, pdfs...), une zone de dessin pour la saisie des modèles et une console pour l'affichage d'informations. L'interface est disponible en français et en anglais.

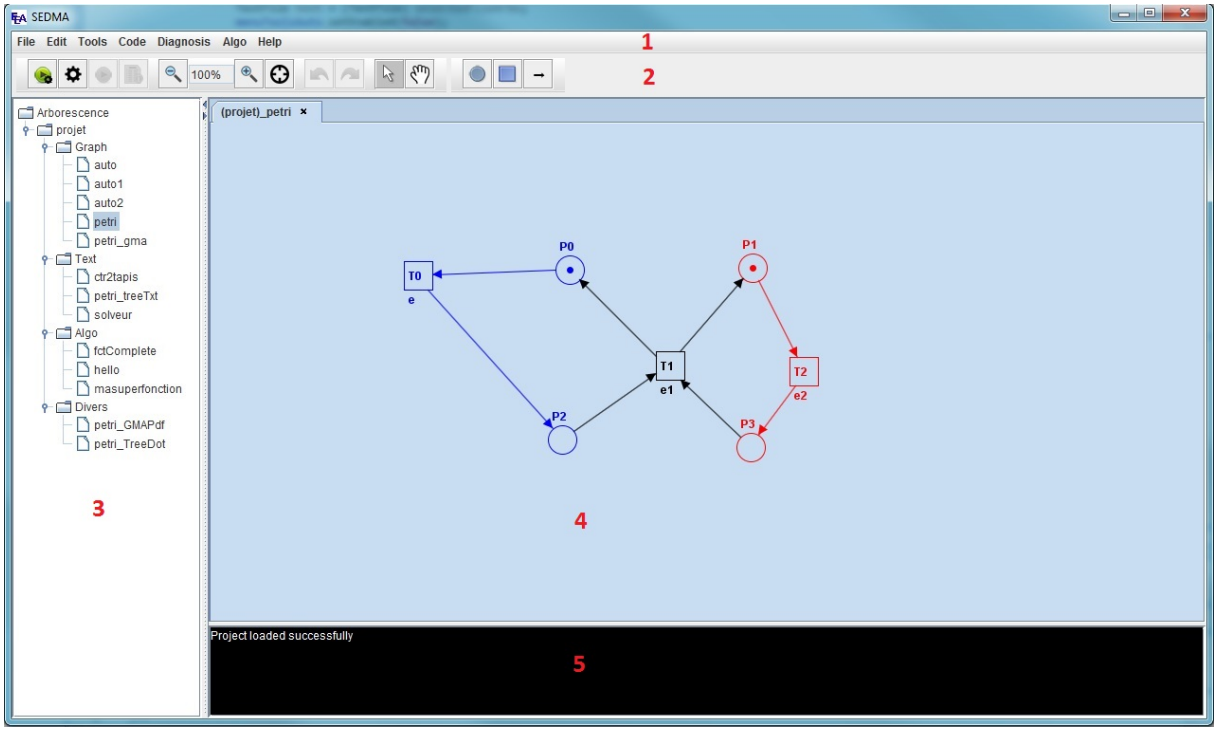


FIGURE 61 – Différentes zones de l'interface graphique.

L'interface graphique contient plusieurs zones distinctes :

- Une barre de menus (1).
- Une barre d'outils (2).
- Une arborescence de projet (3) : l'arborescence peut contenir plusieurs projets, chaque projet peut lui-même contenir plusieurs fichiers (graphes, fichiers textes, pdfs, etc.).
- Une zone de travail (4) : c'est dans cette zone que l'on va pouvoir éditer un graphe ou un fichier texte.
- Une console (5) : différentes informations seront affichées à l'utilisateur.

Barre de menus

Cette zone regroupe la majorité des fonctionnalités de notre logiciel, classées par menus. On retrouve donc :

— « File », comme sur la majorité des logiciels : création, sauvegarde ou ouverture de fichiers ;
— « Edit », fonctionnalités de copier/couper/coller et également « annuler » et « refaire » ;
— « Tools », ce menu regroupe les outils propre à chaque formalisme (produit d'automate, fusion de réseau de Petri, analyse de contraintes, etc.) ;
— « Code », ce menu regroupe les fonctions liées à la génération automatique de programme ;
— « Diagnosis », différents types de diagnostiqueur pour réseau de Petri sont proposés dans ce menu ;
— « Algorithm », ce menu permet de créer et d'exécuter des algorithmes personnalisés ;
— « Help », permet d'accéder à différentes options du logiciel (chemin des librairies externes, changer la langue du logiciel, etc.) ;

Les menus sont dynamiques, c'est à dire qu'en fonction du type de fichier sélectionné (fichier texte, automate, RdP...) certains menus seront désactivés et d'autres seront modifiés.

Barre d'outil

Cette zone permet un accès rapide aux fonctionnalités couramment utilisées. Sont disponibles de gauche à droite :

— des boutons liés à la génération automatique de code ;
— des outils de zoom et le centrage automatique de la vue ;
— les boutons « annuler » et « refaire » ;
— les outils de création de graphes.

Il est à noter également qu'une grande partie des fonctions de la barre de menu et de la barre d'outil sont accessibles via des raccourcis clavier. Ces raccourcis sont précisés sur les boutons via des textes apparaissant lors du survol de la souris pendant quelques secondes.

Arborescence de projet

La partie de gauche est une arborescence de projet, comme on en retrouve sur de nombreux IDE (eclipse, visual studio, etc.). Elle comporte les différents fichiers créés, utilisés ou générés via SEDMA. Cette arborescence comporte 3 niveaux.

1. Au niveau le plus haut on trouve les projets. Un projet regroupe tous les fichiers liés à un travail.
2. Le second niveau comporte des sous-dossiers. Ceux-ci permettent de trier les fichiers selon leurs fonctions.

3. Au dernier niveau se trouvent tous les fichiers créés. Un double clic sur l'un d'eux l'ouvre dans la zone de travail ou le rend actif (le met au premier plan) s'il était déjà ouvert. Si le fichier est dans le dossier « Divers », le logiciel par défaut du système d'exploitation sera utilisé pour l'ouverture hors de SEDMA.

Les fichiers sont interactifs, ont peut donc les renommer, supprimer ou copier/couper/coller via un clic droit ou des raccourcis clavier. Via un clique droit sur un fichier, un menu permettant d'accéder à différentes fonctionnalités est disponible : renommer, copier, exporter, etc.

Enfin, cette arborescence représente également la façon dont le projet va être enregistré sur l'ordinateur. En effet au moment de la sauvegarde un dossier portant le nom du projet et contenant 4 sous-dossiers est créé, dans chacun de ces sous-dossiers sont enregistrés les fichiers apparaissant dans l'arborescence.

Zone de travail

Ceci est la zone principale de SEDMA. C'est ici que sont affichés et édités les fichiers. Il s'agit d'une fenêtre à onglets, on peut donc avoir un fichier par onglet et naviguer entre les onglets ouverts.

L'édition des fichiers se fait via les outils de la barre d'outil ou les raccourcis claviers, à savoir : pour un réseau de Petri une place est créée via un clic sur le bouton central de la souris, une transition avec control+clic bouton central et un arc en maintenant le clic central et en glissant d'une place à une transition (ou l'inverse). Pour un automate on crée un état avec un clic central et un arc en maintenant le clic central et en glissant d'un état à un autre.

Il est ensuite possible d'éditer les caractéristiques de chaque élément via un clic droit puis edit ou un double clic sur l'élément. Un menu proposant divers paramètres s'ouvre alors. On peut également supprimer les éléments via un clic droit puis delete ou via la touche suppr.

Enfin, voici une liste non exhaustive d'autres fonctionnalités également présentes : zoom, copier/couper/coller, mise en couleur de tout ou partie d'un graphe, etc.

II-3 Présentation des fonctionnalités principales

II-3.1 Automates à états finis

SEDMA offre la possibilité de créer graphiquement des automates à états finis et d'appliquer certaines opérations mathématiques et algorithmiques dessus.

Définitions

Deux types d'automates peuvent être utilisés dans SEDMA : les automates de Moore (Moore, 1962) et les automates de Mealy (Mealy, 1955). Les définitions de ces automates sont rappelés ci-dessous.

Définition 39. ***Automate de Moore***
Un automate de Moore est un sextuplet $\mathcal{A} = \langle X, x_0, \Sigma_E, \Sigma_S, \delta, \lambda \rangle$ *avec :*
- X *: ensemble des états discrets*
- x_0 *: état initial,* $x_0 \in X$
- Σ_E *: alphabet d'entrée*
- Σ_S *: alphabet de sortie*
- $\delta : X \times \Sigma_E \to X$ *fonction de transition*
- $\lambda : X \to \Sigma_S$ *fonction de sortie*

Dans un Automate de Moore, les sorties (Σ_S) sont représentées graphiquement par des labels sur les états, et les entrées (Σ_E) par des labels sur les transitions.

Définition 40. ***Automate de Mealy***
Un automate de Mealy est un sextuplet $\mathcal{A} = \langle X, x_0, \Sigma_E, \Sigma_S, \delta, \lambda \rangle$ *avec :*
- X *: ensemble des états discrets*
- x_0 *: état initial,* $x_0 \in X$
- Σ_E *: alphabet d'entrée*
- Σ_S *: alphabet de sortie*
- $\delta : X \times \Sigma_E \to X$ *fonction de transition*
- $\lambda : X \times \Sigma_E \to \Sigma_S$ *fonction de sortie*

Dans un automate de Mealy, les sorties et les entrées sont représentées graphiquement par des labels sur les transitions.

La définition d'un automate dans SEDMA est présentée dans la définition 41. Celle-ci permet d'englober les deux définitions précédentes.

Définition 41. ***Automate à états finis déterministe interprété***
Un automate à états finis déterministe interprété est un septuplet $\mathcal{A} = \langle X, x_0, \Sigma_E, \Sigma_S, \delta, \lambda, X_f \rangle$ *avec :*
- X *: ensemble des états discrets*
- x_0 *: état initial,* $x_0 \in X$
- Σ_E *: alphabet d'entrée*
- Σ_S *: alphabet de sortie*
- $\delta : X \times \Sigma_E \to X$ *fonction de transition*
- $\lambda : X \to \Sigma_S$ *fonction de sortie*
- X_f *: ensemble des états marqués* $X_f \subset X$

Dans un but de synthèse d'une loi de commande, les entrées peuvent correspondre aux capteurs et les sorties aux actionneurs.

La représentation graphique utilisée est illustrée dans la figure 62. Les états sont représentés par des cercles ($\{x0, x1, x2\} \in X$), l'état initial possède une flèche ($x0$), les états terminaux par deux cercles concentriques ($x2 \in X_f$), la fonction de transition est représentée par un arc entre l'état de départ et d'arrivée avec un élément de Σ_E associé ($\{e0, e1\} \in \Sigma_E$), enfin la fonction de sortie est représentée par un élément de Σ_S ($\{s0, s1\} \in \Sigma_S$) attaché à l'état correspondant.

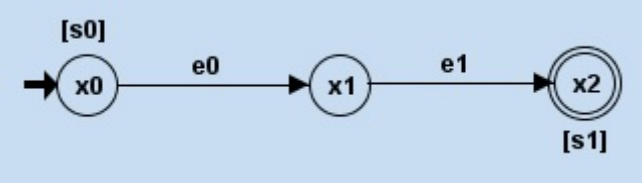


FIGURE 62 – Représentation graphique d'un automate de Moore dans SEDMA

Il est à noter que si $\Sigma_S = \emptyset$, alors la définition 41 devient équivalente à celle des automates déterministes classiques (Déf. 36), les opérations mathématiques standards pourront donc être appliquées.

Opérations disponibles sur les automates

Lorsque qu'un automate est sélectionné dans l'arborescence du projet, différentes fonctionnalités deviennent disponibles dans le menu outils :

Compositions entre deux automates

Le résultat d'une composition de deux automates consiste à créer un nouvel automate en suivant des règles mathématiques. Différentes compositions existent en fonctions des règles mathématiques utilisées. Trois compositions sont disponibles dans SEDMA : la composition synchrone, la composition parallèle et la composition asynchrone (ou produit cartésien). Ces compositions sont tirées du cours de Michel Combacau (Combacau, 2013).

- Composition synchrone : ne fait apparaître que les transitions communes aux deux automates (intersection des alphabets).
- Composition parallèle : elle permet de mettre en évidence les évolutions communes aux deux automates en les synchronisant. Mais en représentant également les évolutions individuelles, c'est à dire les évolutions portant sur des événements exclusifs à un des automates.
- Composition asynchrone : le résultat de cette composition est un automate représentant l'ensemble des combinaisons d'évolutions possibles, que les événements appartiennent ou non à l'intersection des alphabets des automates de départ.

Suppression automatique d'états non-désirés

Dans SEDMA, deux types d'états peuvent être supprimés : les états non accessible et les états non co-accessible. Un état accessible est un état qui peut être rejoint à partir de l'état initial, un état co-accessible est un état à partir duquel on peut rejoindre un état marqué.

Trois fonctions sont disponibles :

— « Accessible » : supprime les états non accessible d'un automate ;
— « Coaccessible » : supprime les états non co-accessible d'un automate ;
— « Trim » : effectue successivement les deux fonctions précédentes.

Il est à noter que le résultat est un automate mathématiques, la représentation graphique (positions des places) n'est pas traitée de façon automatique. Les places de l'automate ainsi créé doivent donc être replacée par la suite. Il est néanmoins possible d'effectuer un placement automatiquement, en effectuant un clique droit dans l'arborescence du projet sur le nouvel automate, et en choisissant la fonction « retracer ». Cette fonction appelle la librairie « graphviz » (Ellson *et al.*, 2004) (fournie avec SEDMA).

Conversion d'un automate en réseau de Petri

Cette fonction permet la création d'un réseau de Petri équivalent à l'automate sélectionné. Lors de l'exécution de cette fonction, un réseau de Petri est créé et stocké dans le presse papier de l'ordinateur. Il suffit de coller le résultat dans le sous-dossier « Graph » d'un projet dans l'arborescence de SEDMA.

Fonctions accessibles depuis l'arborescence

A partir de l'arborescence il est également possible d'accéder à certaines fonctions en effectuant un clique droit sur un graphe :

— importer et d'exporter des automates depuis et vers différents logiciels (TINA, DESUMA, Graphviz, etc.) ;
— exporter le graphe au format image pour prendre en compte les couleurs choisies dans le graphe ;
— enregistrer, sauvegarder, copier, couper, etc. ;
— retracer l'automate en utilisant la librairie Graphviz (Dot).

II-3.2 Réseaux de Petri

Tout comme pour les automates, il est possible de créer des réseaux de Petri.

Définition 42. *Un réseau de Petri ordinaire ($\mathcal{R}$) marqué est un quintuplet $\mathcal{R} = \langle P, T, Pre, Post, M_0 \rangle$ avec :*
- *P : ensemble des places*
- *T : ensemble des transitions, $P \cap T = \emptyset$*
- *$Pre : P \times T \to \mathbb{N}$ fonction d'incidence avant. $Pre(p_i, t_j)$ prend la valeur 1 si et seulement si un arc orienté existe de la place $p_i \in P$ vers la transition $t_j \in T$.*
- *$Post : T \times P \to \mathbb{N}$ fonction d'incidence arrière. $Post(t_j, p_i)$ prend la valeur 1 si et seulement si un arc orienté existe de la transition $t_j \in T$ vers la place $p_i \in P$.*
- *M_0 : marquage initial*

Le formalisme utilisé dans SEDMA est celui des réseaux de Petri labellisés (définition 43).

Définition 43. *Un réseau de Petri labellisé est un sextuplet $\mathcal{R} = \langle P, T, Pre, Post, M_0, E, L \rangle$ avec :*
- *$\langle P, T, Pre, Post, M_0 \rangle$: est un réseau de Petri marqué (définition 42)*
- *$E = Ac \cup Pr$ est un ensemble d'étiquettes désignant des actions (Ac) et des prédicats (Pr)*
- *$L : P \times T \to E$ est une fonction associant des étiquettes aux places ou aux transitions*

Ce formalisme permet de définir les entrées (capteurs) comme des prédicats sur les transitions et les sorties (actionneurs) comme des actions sur les places.

La représentation graphique utilisée est illustrée dans la figure 63. Les places sont représentées par des cercles ($\{P0, P1\}$), les transitions par des carrés ($\{T0, T1\}$), les fonctions Pre et Post par des arcs place/transition ou transition/place, le marquage d'une place est représenté par un jeton ou un nombre, les étiquettes d'actions associées aux places sont entre crochets ([Action1]), enfin les étiquettes de prédicats associées aux transitions sont écrites proches de la transition (capteur1, condition1).

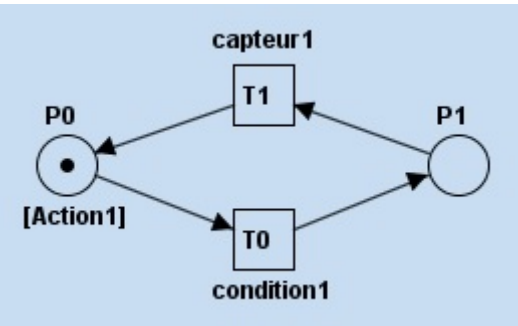


FIGURE 63 – Représentation graphique d'un réseau de Petri dans SEDMA

Opérations disponibles sur les réseaux de Petri

Lorsque qu'un réseau de Petri est sélectionné dans l'arborescence du projet, différentes fonctionnalités deviennent disponibles dans le menu outils :

Fusion de transitions entre réseaux de Petri

Cette opération permet d'effectuer une fusion de transitions sur deux réseaux de Petri. Cette opération est à rapprocher de la composition parallèle des automates. Afin de réaliser une fusion de transition, il faut qu'une ou plusieurs transitions soi(en)t présente(s) dans les deux réseaux, c'est-à-dire qu'elles aient le même nom. Toutes les transitions portant le même nom seront fusionnées. Dans le cas où l'on voudrait fusionner une transition d'un réseau avec plusieurs transitions de l'autre, il suffit de rajouter un ' à la fin de leur nom. Par exemple, la transition T0 du réseau 1 sera fusionnée avec les transitions T0, T0' et T0" du réseau 2.

Attention, les transitions à fusionner ne peuvent pas avoir de prédicats différents. Cependant, il est possible qu'elles aient un prédicat dans un réseau et qu'elles n'en aient pas dans l'autre. Dans ce cas, le résultat aura le prédicat du premier réseau.

Une fois que le réseau résultant de la fusion est créé, l'ensemble des éléments du second graphe est pré-sélectionné, cela facilite leur déplacement pour obtenir un graphe lisible. De plus, pour mieux se repérer, les noms des places ont été préfixés par les trois premières lettres du nom du réseau de Petri d'origine.

Simulateur pas à pas

Cet outil permet de jouer un réseau de Petri et d'observer l'évolution de son marquage après une séquence de franchissement de transitions. Le simulateur est ouvert dans un nouvel onglet du logiciel, dans cet onglet se trouve une version non éditable du réseau de Petri d'origine, avec en rouge toutes les transitions sensibilisées. Pour simuler le franchissement d'une transition, il suffit de cliquer dessus. Le marquage du réseau va donc évoluer en conséquence et de nouvelles transitions seront sensibilisées.

Analyse par énumération des marquages accessibles

Il s'agit de tenter de construire le graphe d'états équivalent au réseau de Petri. Il se nomme le graphe des marquages accessibles. Un sommet du graphe (état) est associé à un marquage, un arc du graphe d'états est associé au prédicat de la transition équivalente dans le réseau de Petri. Avant de pouvoir générer ce graphe il est nécessaire de savoir si le réseau de Petri est borné, sinon le graphe généré sera infini. Cette analyse est faite à l'aide d'un arbre de couverture.

L'arbre de couverture d'un réseau de Petri est représenté par un arbre non binaire, dont les feuilles sont les marquages de ce réseau. La construction de cet arbre se fait pas à pas, c'est à dire que nous partons du marquage initial (racine de l'arbre) et cherchons à franchir toutes les transitions sensibilisées à partir de ce marquage. De ce fait l'arbre s'agrandit à chaque tir de transition, pour chaque nouveau marquage nous cherchons à franchir les transitions sensibilisées depuis ce marquage.

Chaque feuille peut prendre un attribut :

— **DUPLICAT** : signifie que ce marquage existe déjà dans l'arbre, l'exploration de cette branche s'arrête.
— **TERMINAL** : signifie que ce marquage ne sensibilise aucune transition.
— **DIVERGENT** : signifie qu'il existe un marquage, dans la branche reliant cette feuille à la racine, strictement inférieur au marquage de la feuille considérée.

Après avoir calculé l'arbre de couverture nous pouvons former le graphe des marquages accessibles si le réseau est borné, c'est à dire s'il ne contient pas de feuille DIVERGENT. Cela consiste en un re-pliage des feuilles DUPLICAT.

A partir des étiquettes de l'arbre de couverture et du graphe des marquages (GMA) il est possible d'étudier 3 propriétés du réseau de Petri : borné ou non, vivacité, réinitiabilité. L'algorithme 5 résume les étapes suivies et les résultats possibles de l'analyse. Il est à noter qu'il n'est pas toujours possible de conclure sur certaines propriétés.

Algorithme 5 : Algorithme d'analyse de l'arbre de couverture

```
début
    pour chaque feuille de l'arbre faire
        si feuille divergente alors
            ⇒ réseau non borné;
            si feuille terminale alors
                ⇒ réseau non vivant et non réinitialisable;
            sinon
                ⇒ On ne peut pas conclure sur le caractère vivant et réinitialisable;
            fin
        sinon
            ⇒ réseau borné;
            si feuille terminale alors
                ⇒ réseau non vivant et non réinitialisable;
            sinon
                si tout le GMA est UNE composante fortement connexe alors
                    ⇒ réseau réinitialisable ;
                sinon
                    ⇒ pas de conclusion sur réinitilisable;
                    si toutes les transitions sont dans les composantes fortement
                     connexes alors
                        ⇒ réseau vivant;
                    sinon
                        ⇒ pas de conclusion sur la vivacité;
                    fin
                fin
            fin
        fin
    fin
fin
```

L'arbre de couverture et le graphe des marquages peuvent être enregistrés dans le projet afin d'être affiché. Si la librairie Graphviz (Ellson *et al.*, 2004) est disponible, elle est utilisée

pour le placement des nœuds. Sinon un affichage moins optimal est automatiquement proposé.

Analyse structurelle

Il est également possible d'analyser le réseau via une étude structurelle. Celle ci est basée sur le calcul et l'étude des invariants de places et de transitions du réseau de Petri. La librairie *struct* de l'outil TINA (Berthomieu *et al.*, 2004) est automatiquement inclue et appelée dans SEDMA. Le résultat est un fichier texte contenant l'analyse et est affiché dans SEDMA dans la zone de travail.

Enfin, tout comme pour les automates, il est possible d'importer et d'exporter depuis et vers les logiciels TINA (Berthomieu *et al.*, 2004) et au format .dot de Graphviz (Ellson *et al.*, 2004). Il est également possible d'enregistrer le graphe au format image.

II-3.3 Filtre logique

Dès qu'un fichier texte est sélectionné dans l'arborescence, il est possible d'accéder à des fonctions via le menu « outils » permettant de travailler sur des contraintes logiques telles que définies dans ce mémoire de thèse.

Format d'entrée

Le fichier d'entrée est formaté à l'aide de mots clefs en début de ligne permettant d'identifier ce qui va-t-être défini dans la suite de la ligne. Le format est décrit ci-dessous, les parenthèses signifient que l'élément est optionnel, « * » signifie qu'il peut être répété, « | » signifie « ou ».

- Commentaire
  // <commentaire>
- Description des actionneurs
  a <nom de la variable> (, <nom de la variable>)*
- Description des capteurs
  c <nom de la variable> (, <nom de la variable>)*
- Description des observateurs
  o <nom de la variable> (= <équation logique>)
- Description des contraintes simples
  css <équation logique>
- Description des contraintes combinées
  csc <équation logique> ; F_ :(<actionneur à forcer> (, <actionneur à forcer>)) (;
  F_<actionneur forcé> :(<actionneur à forcer> (, <actionneur à forcer>)*))*
- Description des contraintes structurelles
  struct <équation logique>
- Langage cible (C# ou ST), par défaut « st »
  config cs | st

Un exemple complet minimal est donné figure 64.

```
// exemple de fichier
// Description des variables commandables (actionneurs)
a A1, A2
a A3
// Description des variables non commandables et des observateurs
c c1, c2, c3
o obs1 = c1.c2
o obs2
// Contraintes simples
css c1.A1
css !c1.!A2
css c3.A1*.A2
// Contraintes combinées
csc !A1.A2 ; F_:(A1)
csc A1.A2.A3 ; F_:(A1,A2) F_A1:(A3) F_A2:(A1,A3) F_A3:(A1)
// Contraintes structurelles et configuration
struct c1.c3
config st
```

FIGURE 64 – Exemple de fichier décrivant les contraintes logiques

Les notations utilisées sont les suivantes : ! pour le complément logique, . pour le ET logique, + pour le OU logique, $*$ pour indiquer la mémoire ($A1^*$ est la valeur de $A1$ au cycle précédent).

Les priorités fonctionnelles de l'exemple précédent sont les suivantes :

- « csc A1.A2 »
  - « F_ :(A1) » : forcer A1 à 1 si aucune variable commandable n'est forcée par des contraintes simples et que la contrainte est violée ;
- « csc A1.A2.A3 »
  - « F_ :(A1,A2) » : forcer A1 et A2 à 0 si aucune variable commandable n'est forcée par des contraintes simples et que la contrainte est violée ;
  - « F_A1 :(A3) » : forcer A3 à 0 si A1 est forcée à 1 par des contraintes simples et que la contrainte est violée ;
  - « F_A2 :(A1,A3) » : forcer A1 et A3 à 0 si A2 à 1 est forcée par des contraintes simples et que la contrainte est violée ;
  - « F_A3 :(A1) » : forcer A1 à 0 si A3 est forcée à 1 par des contraintes simples et que la contrainte est violée ;

Pour un fichier de contrainte donné, il est possible d'effectuer différentes opérations présentées ci-après.

Analyse de la cohérence par la méthode des graphes

Cette fonction permet d'appliquer l'approche détaillée précédemment dans ce mémoire pour la vérification de la cohérence (section III-4.3).

Génération de l'algorithme

L'algorithme est généré en langage ST (Structured Text) ou en C#, il est ensuite ajouté

au projet actif.

Génération du fichier de configuration

Un fichier au format .xsy est généré (automate Schneider) permettant la configuration automatique des variables nécessaires à l'exécution du filtre logique dans l'automate.

Génération du fichier BESS

Un fichier au format BESS est généré afin d'utiliser cet outil pour l'analyse de la cohérence algébrique comme détaillé précédemment dans ce mémoire (section III-4.3).

II-3.4 Diagnostic

Actuellement SEDMA permet d'utiliser des outils de diagnostic uniquement sur des réseaux de Petri. Cependant, l'outil de conversion d'un automate en réseau de Petri permet l'application indirect de ces méthodes de diagnostic sur des automates.

Configuration des événements

La première étape pour utiliser ces fonctions de diagnostic est de définir la liste des événements observables et fautifs. Pour cela, il est nécessaire d'ouvrir le menu accessible via le bouton « définir événements » du menu « diagnostic ». Ce menu affiche une liste de tous les événements utilisés dans le réseau de Petri. Il faut alors cocher les cases correspondantes pour indiquer si cet événement est observable ou non et fautif ou non. Une fois la validation faite et si aucune couleur n'avait été appliquée au graphe, les événements inobservables sont grisés tandis quand les événements fautifs sont indiqués en rouge.

Diagnostiqueur de réseau de Petri

Une fois les événements configurés, il est possible de calculer différents types de diagnostiqueurs. Les différentes versions sont basées sur le diagnostiqueur de Sampath (Sampath *et al.*, 1996).

Diagnostiqueur monolithique

Le diagnostiqueur classique de Sampath, appelé monolithique, peut-être calculé ainsi que sa version coloriée (Pencolé *et al.*, 2015). La version coloriée permet une représentation graphique compacte à l'aide du formalisme des réseaux de Petri coloré. Une option de calcul est disponible appelée « clôture silencieuse », celle-ci permet un enrichissement des états de croyances du diagnostiqueur (Zanella et Lamperti, 2003).

Diagnostiqueur modulaire

Une approche de diagnostic modulaire basée sur Pencolé *et al.* (2015) est proposée dans SEDMA. Il est possible de définir, dans un même réseau de Petri, différents modules. A l'exécution de cette fonctionnalité, le nombre de modules à définir est demandé, puis une

fenêtre permettant de décrire quelles places et quelles transitions appartiennent à quels modules. Si des couleurs avaient été attribuées aux éléments du réseaux, les modules sont prédéfinis automatiquement mais il est toujours possible de les modifier via la fenêtre de configuration. L'exemple de réseau de Petri de la figure 61 présente deux modules : {P0, T0, P2, T1, bleu} et {P1, T2, P3, T1, rouge}, la transition T1 est en noir car l'événement associé (e1) est partagé entre les deux modules.

Après la configuration des modules et des événements, il est possible de calculer le diagnostiqueur de Sampath pour chaque module (diagnostiqueurs locaux), le diagnostiqueur modulaire, sa version colorée (Pencolé *et al.*, 2015) ainsi que la composition parallèle des diagnostiqueurs locaux. Il est à noter que pour avoir l'équivalence entre le diagnostiqueur de Sampath monolithique et la composition parallèle des diagnostiqueurs locaux, il est nécessaire d'utiliser l'option « clôture silencieuse » pour les calculs.

II-3.5 Génération automatique de programme API

Il est possible, à partir d'un automate ou d'un réseau de Petri, de générer le programme correspondant dans différents langages : le C, le ST et le VHDL. En ce qui concerne l'approche par filtre logique, il est possible de générer l'algorithme en ST ou en C#, ainsi que le fichier de déclaration de variables pour les automates Schneider.

II-3.6 Algorithme personnalisé

SEDMA offre la possibilité à l'utilisateur d'exécuter ses propres fonctions sur les automates ou les réseaux de Petri. La création d'une fonction est accessible depuis le menu « Algorithme » ainsi que par un clique droit sur le dossier « Algo » dans l'arborescence. Une fonction est définie par un nom, un type de retour (Automata, PetriNetwork ou void), un ou des paramètres d'entrée (nom et type) ainsi qu'un code devant être écrit dans le langage JAVA. La figure 65 présente la création d'une fonction avec un paramètre de sortie de type réseau de Petri et un paramètre d'entrée de type réseau de Petri également.

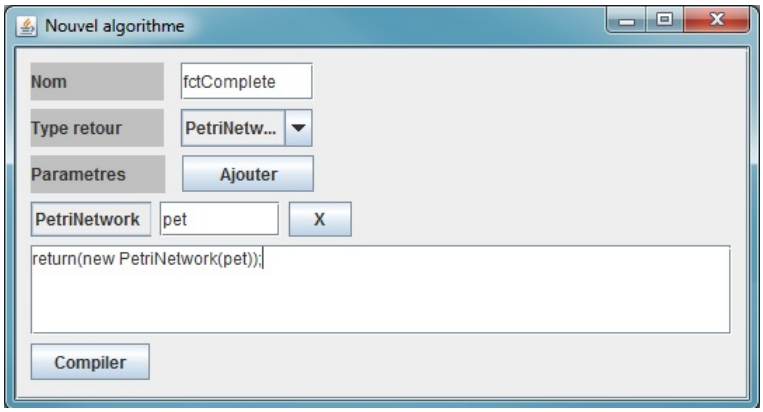


FIGURE 65 – Exemple de création d'un algorithme personnalisé

Une fois l'algorithme créé, il peut être exécuté depuis le menu « Algorithme » ou via un double clique dans l'arborescence du projet. Si la fonction contient un type de retour, le graphe créé sera rajouté au projet contenant l'algorithme utilisé et un nom sera demandé.

Dans un cadre pédagogique cette fonctionnalité permet de fournir un environnement unique pour l'apprentissage de la programmation et l'application sur des graphes (parcours, analyse, placement de noeuds, etc.). D'un point de vue scientifique il est aisé de comparer différents algorithmes traitant le même problème.

II-4 Conclusion

SEDMA est un outil logiciel permettant la modélisation, l'analyse et la génération automatique de programme pour les systèmes à événements discrets. SEDMA a pour but de fournir un environnement unique pour l'enseignement et l'application des systèmes à événements discrets au travers de plusieurs formalismes de modélisation (automates à états finis, réseaux de Petri, contraintes logiques). Pour chaque formalisme des outils mathématiques et algorithmiques sont proposés, ceux-ci permettent la création ou bien l'analyse des modèles étudiés.

Dans le cadre de cette thèse, une restructuration du logiciel a été apportée. Cette restructuration permet d'intégrer plus aisément de nouvelles fonctionnalités. L'ensemble des algorithmes liés au contraintes logiques et à l'approche par filtre logique développé dans cette thèse ont été implémenté dans SEDMA.

A l'aide de SEDMA, il est à présent possible d'obtenir un filtre logique cohérent, implémenté en ST, de façon automatique à partir d'une description textuelle des contraintes logiques. Cet outil facilite l'utilisation de la méthode de conception de contrôleur proposée dans cette thèse.

ANNEXE C

Expérimentation Scratch2 et HOME I/O

Sujet donné au groupe A

EXPERIMENTATION : Méthodologie de programmation d'un portail automatique

Référents du projet :

Romain PICHARD (romain.pichard@univ-reims.fr)

Alexandre PHILIPPOT (alexandre.philippot@univ-reims.fr)

Description du projet

Le but de cette expérimentation est de concevoir un programme de contrôle/commande permettant la gestion automatique d'un portail. Le cahier des charges du programme est décrit ci-dessous.

Un document Word présentant votre méthodologie et votre solution doit être fourni par email à la fin du projet.

Vous êtes totalement libre sur l'organisation de votre travail : méthode à utiliser, règle de programmation… Néanmoins, votre méthodologie, si vous en avez utilisé une, devra être présentée dans votre livrable.

Contraintes techniques

La programmation doit être effectuée sur l'environnement Scratch 2.

Le logiciel HOME I/O doit être utilisé pour réaliser vos tests de vérification et de validation.

Un guide d'utilisation rapide de HOME I/O avec Scratch 2 est disponible à l'adresse suivante :

https://realgames.co/docs/homeio/fr/scratch2/

Plusieurs actions sont possibles sur le portail :

- ouvrir : permet d'actionner le moteur dans le sens de l'ouverture
- fermer : permet d'actionner le moteur dans le sens de la fermeture
- stopper : permet d'arrêter le moteur

Des capteurs détectant l'ouverture et la fermeture du portail sont disponibles :

- ouvert : le portail est complètement ouvert
- fermé : le portail est complètement fermé

L'état du bouton de la télécommande (bp1) peut être récupéré comme indiqué ci-dessous :

Cahier des charges

- Lorsque le portail est fermé :
    - 1 appui sur bp1 ouvre le portail
- Lorsque le portail est en train de s'ouvrir :
    - 1 appui sur bp1 stoppe le portail
    - 1 nouvel appui sur bp1 ferme le portail
- Une fois ouvert, le portail se referme automatiquement après 5 secondes :
    - Si on appuie sur bp1 quand le portail est ouvert, celui-ci se referme immédiatement
- Lorsque le portail est en train de se fermer :
    - 1 appui sur le bp1 stoppe le portail
    - 1 nouvel appui sur bp1 ouvre le portail
- Le moteur doit s'arrêter lorsque le portail est ouvert ou fermé

Il n'est pas demandé de gérer la détection de personnes et/ou de véhicules.

(bp1 fait référence au bouton numéro 1 de la télécommande de HOME I/O)

Livrable :

A la fin de ce projet, vous devez rendre par email un document Word permettant de répondre aux questions suivantes :

- Avez-vous utilisé une ou plusieurs **méthodes** permettant de résoudre ce problème de commande ?
    - Si oui, présentez ces méthodes (schéma, organigramme, explications textuelles …).
- Aviez-vous déjà utilisé Scratch et/ou HOME I/O ?
- Quelles sont les idées principales du fonctionnement et de l'organisation de votre programme ?
- Avez-vous appliqué une méthode permettant la vérification de votre programme ?
    - Si oui, présentez cette méthode (vérification formelle, model-checking, tests unitaires…).
- Combien de temps estimez-vous avoir passé sur :
    - L'analyse du problème ?
    - La programmation sous Scratch ?
- Pensez-vous que votre programme réponde entièrement au cahier des charges ?
- Quelle(s) difficulté(s) avez-vous rencontré ?

ANNEXE D

Filtre logique à base de solveur SAT

IV-1 Code Python pour la preuve de concept

```
import standard_lib
from standard_lib import R_TRIG, F_TRIG, TON
import constraint
from constraint import *

#Variables and function blocks instances
Mem_0=False
Mem_1=False
Mem_2=False
Mem_3=False
Mem_4=False
Mem_5=False
Mem_6=False
Mem_7=False
P36 = False
P67=False
P79 = False
P810 = False
P2=False
PC=False
R_I_0 = R_TRIG()
F_I_0 = F_TRIG()
R_I_1 = R_TRIG()
F_I_1 = F_TRIG()
R_I_2 = R_TRIG()
F_I_2 = F_TRIG()
R_I_3 = R_TRIG()
F_I_3 = F_TRIG()
R_I_4 = R_TRIG()
F_I_4 = F_TRIG()
R_I_5 = R_TRIG()
F_I_5 = F_TRIG()
R_I_6 = R_TRIG()
F_I_6 = F_TRIG()
R_I_7 = R_TRIG()
F_I_7 = F_TRIG()
R_I_8 = R_TRIG()
```

```
F_I_8  =  F_TRIG()
R_I_9  =  R_TRIG()
F_I_9  =  F_TRIG()
R_I_10  =  R_TRIG()
F_I_10  =  F_TRIG()
R_I_11  =  R_TRIG()
F_I_11  =  F_TRIG()
R_I_12  =  R_TRIG()
F_I_12  =  F_TRIG()
R_I_13  =  R_TRIG()
F_I_13  =  F_TRIG()
R_I_14  =  R_TRIG()
F_I_14  =  F_TRIG()
R_I_15  =  R_TRIG()
F_I_15  =  F_TRIG()

x1=False
x2=False
x3=False
x11=False
x12=False
x13=False
x14=False
x15=False
x21=False
x22=False
x31=False
x32=False

Compteur = 0
Compteur1 = 0

def Hamming(x,y):
    compt=0
    for k,v in x.iteritems():
        if y[k] <> v:
            compt = compt+1
    return(compt)

def Hamming_min(prop,solutions):
    best = min(solutions,key = lambda sol:Hamming(prop,sol))
    return(best)

def M1T2(I, O):
    global Mem_0, Mem_1, Mem_2, Mem_3, Mem_4, Mem_5, Mem_6, Mem_7
    global R_I_11, F_I_11,R_I_12, F_I_12, R_I_13, F_I_13, R_I_14, F_I_14,
        R_I_15, F_I_15
    global R_I_0, F_I_0, R_I_1, F_I_1,R_I_2, F_I_2, R_I_3, F_I_3, R_I_4,
        F_I_4, R_I_5, F_I_5, R_I_6, F_I_6,R_I_7, F_I_7, R_I_8, F_I_8, R_I_9,
         F_I_9, R_I_10, F_I_10
    global PC, P36, P67, P2, P79, P810
    global x1, x2, x3
    global x10, x11, x12, x13, x14, x15, x21, x22, x31, x32
    global PC, P36, P67, P2, P79, P810, Compteur, Compteur1

    R_I_11(I[11])
    F_I_11(I[11])
```

```
R_I_12(I[12])
F_I_12(I[12])
R_I_13(I[13])
F_I_13(I[13])
R_I_14(I[14])
F_I_14(I[14])
R_I_15(I[15])
F_I_15(I[15])
R_I_1(I[1])
F_I_1(I[1])
R_I_2(I[2])
F_I_2(I[2])
R_I_3(I[3])
F_I_3(I[3])
R_I_4(I[4])
F_I_4(I[4])
R_I_5(I[5])
F_I_5(I[5])
R_I_6(I[6])
F_I_6(I[6])
R_I_7(I[7])
F_I_7(I[7])
R_I_8(I[8])
F_I_8(I[8])
R_I_9(I[9])
F_I_9(I[9])
R_I_10(I[10])
F_I_10(I[10])
R_I_0(I[0])
F_I_0(I[0])

######################
#Grafcet fonctionnel
######################

#Modes de Marches
if not I[15] :

    x1=False
    x2=False
    x3=False

    x11=False
    x12=False
    x13=False
    x14=False
    x15=False

    x21=False
    x22=False

    x31=False
    x32=False

if I[15] and R_I_14.Q :

    x1=True
```

```
    x2=False
    x3=False

    x11=True
    x12=False
    x13=False
    x14=False
    x15=False

    x21=True
    x22=False

    x31=True
    x32=False
    P67=False
    P36 = False
    P2=False
    PC=False
    P810 = False
    P79 = False
    Compteur = 0


#Grafcet Start / Stop
ft1=x1 and R_I_12.Q
ft2=x2 and F_I_13.Q
ft3= x3 and (Compteur==0) and x11 and x21 and x31
x1= ft3 or (x1 and not ft1)
x2= ft1 or (x2 and not ft2)
x3 = ft2 or (x3 and not ft3)

#Grafcet convoyeur evacuation droite
ft21=x21 and R_I_7.Q and not x1
ft22=x22 and F_I_9.Q
x21= ft22 or (x21 and not ft21)
x22= ft21 or (x22 and not ft22)

#Grafcet convoyeur evacuation gauche
ft31=x31 and R_I_8.Q and not x1
ft32=x32 and F_I_10.Q
x31= ft32 or (x31 and not ft31)
x32= ft31 or (x32 and not ft32)

#Grafcet plateau tournant
ft11= x11 and not x1 and I[3]
ft12=x12 and I[6]
ft13_1=x13 and I[5] and PC
ft13_2=x13 and I[5] and not PC
ft14=x14 and F_I_8.Q
ft15=x15 and F_I_7.Q
x11=ft14 or ft15 or (x11 and not ft11)
x12=ft11 or (x12 and not ft12)
x13=ft12 or (x13 and not (ft13_1 or ft13_2))
x14=ft13_1 or (x14 and not ft14)
x15=ft13_2 or (x15 and not ft15)

######################
```

```
#Observateurs logiques
######################

if F_I_7.Q or F_I_8.Q :
    PC = not PC

if F_I_9.Q or F_I_10.Q :
    Compteur1 = Compteur1 +1
    print "nbre de caisses : ",Compteur1

if I[1] and R_I_0.Q :
    P2 = True
if not I[1] or not I[0] :
    P2 = False

if I[4] and F_I_3.Q :
    P36 = True

if P36 and I[4] and R_I_6.Q :
    P36 = False

if I[5] and F_I_6.Q and not I[8] :
    P67 = True

if P67 and (R_I_6.Q or R_I_7.Q) :
    P67 = False

if F_I_7.Q :
    P79 = True
if R_I_9.Q :
    P79= False

if F_I_8.Q :
    P810=True
if R_I_10.Q :
    P810 = False

if F_I_0.Q :
    Compteur = Compteur+1
if F_I_3.Q :
    Compteur = Compteur-1

######################
#Fonctions de forçages simples
######################

F0s0 = P2
F1s0 = False

F0s1 = (I[3] and not I[4]) or (I[3]and I[4] and I[6]) or (I[3] and P36)
F1s1 = False

F0s2 = (I[4] and I[6]) or (not I[4] and not I[5]) or (I[5] and I[7])
F1s2 = False

F0s3 = not I[5]
F1s3 = False
```

```
F0s4 = (I[4] and not I[6]) or (I[4] and (P810 or I[10] or I[9] or P79))
F1s4 = (I[5] and I[6]) or I[8] or I[7] or P67

F0s5= False
F1s5 = False

F0s6 = False
F1s6 = False

#Vecteur des sorties souhaitées

g10 = x2 or (x3 and I[0])
g11 = (x2 and (I[0] or Compteur<>0))  or (x3 and Compteur<>0)
g12 = x12 or x14
g13 = x15
g14 = x13
g15 = x32
g16 = x22
fov = {"a":g10,"b":g11,"c":g12,"d":g13,"e":g14,"f":g15,"g":g16}

problem = Problem()
problem.addVariables(["a","b","c","d","e","f","g"],[False,True])

problem.addConstraint(lambda a : (a and F0s0 or not a and F1s0)==False
    ,('a'))
problem.addConstraint(lambda b : (b and F0s1 or not b and F1s1)==False
    ,('b'))
problem.addConstraint(lambda c : (c and F0s2 or not c and F1s2)==False
    ,('c'))
problem.addConstraint(lambda d : (d and F0s3 or not d and F1s3)==False
    ,('d'))
problem.addConstraint(lambda e : (e and F0s4 or not e and F1s4)==False
    ,('e'))
problem.addConstraint(lambda f : (f and F0s5 or not f and F1s5)==False
    ,('f'))
problem.addConstraint(lambda g : (g and F0s6 or not g and F1s6)==False
    ,('g'))

problem.addConstraint(lambda a,b : (a and not b and I[0])==False,('a','
    b'))
problem.addConstraint(lambda b,c : (b and not c and I[3] and I[4])==
    False,('b','c'))
problem.addConstraint(lambda c,f : (c and not f and I[5] and I[8])==
    False,('c','f'))
problem.addConstraint(lambda d,g : (d and not g and I[5] and I[7])==
    False,('d','g'))
problem.addConstraint(lambda c,d : (c and d)==False,('c','d'))

#Calcul de l'ensemble des solutions
solutions = problem.getSolutions()

print len(solutions),"␣solutions",

#Choix de la solution
sol = Hamming_min(fov, solutions)
```

```
    O[0] = sol.get("a")
    O[1] = sol.get("b")
    O[2] = sol.get("c")
    O[3] = sol.get("d")
    O[4] = sol.get("e")
    O[5] = sol.get("f")
    O[6] = sol.get("g")

    problem.reset()

    return
```

IV-2 Implémentation ST de l'algorithme à base de distance de Hamming

```
(* Grafcet fonctionnel et calcul des observateurs logiques *)

IF (RE(Run)) THEN
    P2 := false;
    P36 := false;
    P67 := false;
    P79 := false;
    P810 := false;
END_IF;

P2 := (C1 and RE(C0)) OR P2 AND not (not C1 or not C0);
P36 := (C4 and FE(C3)) OR P36 AND NOT (P36 and C4 and RE(C6));
P67 := (C5 and FE(C6) and not C8) OR P67 AND NOT (P67 and (RE(C6) or RE(C7)
   ));
P79 := (FE(C7)) OR P79 AND NOT (RE(C9));
P810 := (FE(C8)) OR P810 AND NOT (RE(C10));

if (FE(C0)) then
    cpt_conv1 := cpt_conv1+1;
end_if;
if (FE(C3)) then
    cpt_conv1 := cpt_conv1-1;
end_if;
if (X1.x) then
    cpt_conv1 := 0;
    n := 0;
end_if;

(* gestion gauche droite *)

if (X43.x) then
    left[n] := false;
    n := n + 1;
end_if;

if (X44.x) then
    left[n] := true;
    n := n + 1;
end_if;
```

```
if (FE(C8) OR FE(C7)) then
    FOR m := 0 TO 9 DO
        left[m] := left[m+1];
    END_for;
    n := n - 1;
end_if;

(* Affectation des sorties à partir des étapes Grafcet *)

S0:= X2.x or (X3.x and c0);
S1:= (X2.x and (c0 or cpt_conv1<>0)) or (X3.x and cpt_conv1<>0);
S2:= X12.x or X14.x;
S3:= X15.x;
S4:= X13.x;
S5:= X32.x;
S6:= X22.x;

No := 7;
Ncsc := 5;

(* Affectation des Gk à partir des sorties *)
G[0] := S0;
G[1] := S1;
G[2] := S2;
G[3] := S3;
G[4] := S4;
G[5] := S5;
G[6] := S6;

(* Création des F0s à partir des Cs *)
F0s[0] := (P2);
F0s[1] := (C3 AND NOT C4) OR (C3 AND C4 AND C6) OR (C3 AND P36);
F0s[2] := (C4 AND C6) OR (NOT C4 AND NOT C5) OR (C5 AND C7);
F0s[3] := (NOT C5);
F0s[4] := (C4 AND C9) OR (C4 AND NOT C6) OR (C4 AND P79) OR (C4 AND P810)
   OR (C4 AND C10);
F0s[5] := false;
F0s[6] := false;

(* Création des F1s à partir des Cs *)
F1s[0] := false;
F1s[1] := false;
F1s[2] := false;
F1s[3] := false;
F1s[4] := (C5 AND C6) OR (C5 AND C8) OR (C5 AND C7) OR (C5 AND P67);
F1s[5] := false;
F1s[6] := false;

nbr := 0;

(* Initialisation *)
FOR k := 0 TO (No-1) DO
    mem[k] := NOT F0s[k] AND G[k] OR F1s[k];
    membis1[k]:= NOT F0s[k] AND not G[k] OR F1s[k];
END_FOR;
```

```
tabmot:=mem;

calcul_CSC();
test_CSC();

IF Flag THEN
    Flagbis:=true;
    initCSC();

    Noc := 0;
    FOR i := 0 TO (No-1) DO
        IF NOT F0s[i] AND NOT F1s[i] and GG[i] THEN
            tabMot2[Noc]:=i;
            Noc:=Noc+1;
        END_IF;
    END_FOR;

    maxHam:=Noc;

    FOR k := 1 to DIV(Noc,2) do
        Flag1:=true;
        FOR ii := 0 TO k-1 DO (* First k-subset *)
            HamMot[ii]:=True;
        END_FOR;
        fOR ii := k TO Noc-1 DO
            HamMot[ii]:=False;
        END_FOR;

        FOR i := 0 TO (Noc-1) DO  (* test first k-subset *)
            IF HamMot[i] THEN
                mem[tabMot2[i]]:=not tabMot[tabMot2[i]];
            ELSE
                mem[tabMot2[i]]:=tabMot[tabMot2[i]];
            END_IF;
        END_FOR;

        calcul_CSC();
        test_CSC();

        IF Flag and (maxHam>Noc-k) THEN
            FOR i := 0 TO (Noc-1) DO  (* test first k-subset *)
                IF not HamMot[i] THEN
                    mem[tabMot2[i]]:=tabMot[tabMot2[i]];
                ELSE
                    mem[tabMot2[i]]:=not tabMot[tabMot2[i]];
                END_IF;
            END_FOR;

            calcul_CSC();
            test_CSC(); (* bis*)

            IF not Flag THEN
                Flagbis:=False;
                membis1:=mem;
                maxHam:=Noc-k;
                Flag:=True;
            END_IF;
```

```
REPEAT  (* Next k-subset *)
     cpt := 0;
     For i:=0 to Noc-2 do
          IF HamMot[i] THEN
               cpt:=cpt+1;
               IF not HamMot[i+1] THEN
                    EXIT;
               END_IF;
          END_IF;
     END_FOR;

     IF i=Noc-1 THEN
          flag1:=False;
     ELSE
          HamMot[i]:=false;
          HamMot[i+1]:=true;
          fOR j := 0 TO i-1 DO
               HamMot[j]:=false;
          END_FOR;

          WHILE (cpt>1) DO
               HamMot[cpt-2]:=true;
                    cpt:=cpt-1;
          END_WHILE;

          FOR i := 0 TO (Noc-1) DO
               IF HamMot[i] THEN
                    mem[tabMot2[i]]:=not tabMot[tabMot2[i]];
               ELSE
                    mem[tabMot2[i]]:=tabMot[tabMot2[i]];
               END_IF;
          END_FOR;

          calcul_CSC();
          test_CSC();

          IF Flag and (maxHam>Noc-k) THEN
               FOR i := 0 TO (Noc-1) DO
                    IF not HamMot[i] THEN
                         mem[tabMot2[i]]:=not tabMot[tabMot2[i]];
                    ELSE
                         mem[tabMot2[i]]:=tabMot[tabMot2[i]];
                    END_IF;
               END_FOR;

               calcul_CSC();
               test_CSC(); (* bis*)

               IF not Flag THEN
                    Flagbis:=False;
                    membis1:=mem;
                    maxHam:=Noc-k;
                    Flag:=True;
               END_IF;

          END_IF;
```

```
                END_IF;

            UNTIL not flag or not flag1 END_REPEAT;

        END_IF;

        if not Flag Then
            exit;
        end_if;
    END_FOR;

    IF Flag THEN
        IF not Flagbis THEN
            mem:=membis1;
        ELSE
            For i:= 0 to (Noc-1) DO
                mem[tabMot2[i]]:=not tabMot[tabMot2[i]];
            END_FOR;

            calcul_CSC();
            test_CSC(); (* bis *)

            IF Flag THEN
                FOR i := 0 TO (No-1) DO
                    mem[i] := NOT F0s[i] AND F1s[i];
                END_FOR;
            END_IF;
        END_IF;

    End_If;

END_IF;

(* Affectation des sorties finales*)
S0 := mem[0];
S1 := mem[1];
S2 := mem[2];
S3 := mem[3];
S4 := mem[4];
S5 := mem[5];
S6 := mem[6];

##########################
test_CSC(){
Flag := false;

FOR ijk:=0 TO (Ncsc-1) DO
    IF  CSc[ijk] THEN
        Flag:=True;
        EXIT;
    END_IF;(* Mise à jour du flag *)
END_FOR;
}
##########################
```

```
#########################
init_CSC(){
FOR ijk := 0 TO (No-1) DO
    GG[ijk]:=false;
END_FOR;

IF CSC[0] THEN
    GG[0]:=True;
    GG[1]:=true;
END_IF;

IF CSC[1] THEN
    GG[1]:=True;
    GG[2]:=true;
END_IF;

IF CSC[2] THEN
    GG[2]:=True;
    GG[5]:=true;
END_IF;
IF CSC[3] THEN
    GG[3]:=True;
    GG[6]:=true;
END_IF;
IF CSC[4] THEN
    GG[2]:=True;
    GG[3]:=true;
END_IF;
}
#########################

#########################
calcul_CSC(){
CSc[0] := C0 AND mem[0] AND NOT mem[1];
CSc[1] := C3 AND C4 AND mem[1] AND NOT mem[2];
CSc[2] := C5 AND C8 AND mem[2] AND NOT mem[5];
CSc[3] := C5 AND C7 AND mem[3] AND NOT mem[6];
CSc[4] := mem[2] AND mem[3];
}
#########################
```

IV-3 Implémentation ST d'un solveur SAT

IV-3.1 Initialisation du solveur et de la structure de donnée

```
UNDEF := -1;
UNSAT := 0;
SAT := 1;

MAX_NUM_VARIABLE := 7;
MAX_NUM_CLAUSE := 100;
MAX_CLAUSE_SZ := MAX_NUM_VARIABLE + 3; (* Max size of a clause. The "+3" is
    needed to stock some information about the clause. *)
```

```
IS_CL := MAX_NUM_VARIABLE; (* IS_CL to manage clause satifiability *)
ST_CL := MAX_NUM_VARIABLE + 1; (* ST_CL to manage clause assignation *)
SZ_CL := MAX_NUM_VARIABLE + 2; (* SZ_CL MAX_NUM_VARIABLE+2 *)

nb_decisions := 0;
nb_solutions := 0;
formula_status := UNDEF;
picked := 0;

tmp_var := 0;

_decisions := 0;
_current_assignment := 0;
_to_set := 0;

var_to_set := 0;
var_to_unset := 0;

FOR j := 0 TO (MAX_NUM_CLAUSE-1) DO
    MOVE_INT_ARINT(0, formula[j]);
END_FOR;

FOR j := 0 TO 60 DO
    MOVE_INT_ARINT(0, var_in_clause[j]);
END_FOR;
```

IV-3.2 Ajouts des contraintes à la structure de donnée

```
nb_clauses   := 0;
nb_literals  := 7;
nb_outputs := 7 ;

i := 0;

(* prise en compte des combinées *)

(* -1 2 -8 0 <=> S0.!S1.C0*)
IF C0 THEN
    nb_clauses := nb_clauses + 1;
    tmp_var := -1;
    formula[i][0] := tmp_var;
    var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
       MAX_NUM_VARIABLE][0] + 1;
    var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
       MAX_NUM_VARIABLE][0]] := i;

    tmp_var := 2;
    formula[i][1] := tmp_var;
    var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
       MAX_NUM_VARIABLE][0] + 1;
    var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
       MAX_NUM_VARIABLE][0]] := i;

    formula[i][IS_CL] := 0;
    formula[i][ST_CL] := 2;
    formula[i][SZ_CL] := 2;
END_IF;
```

```
(* -2 3 -9 -10 0 <=> S1.!S2.C3.C4 *)
IF C3 AND C4 THEN
i := i + 1;
nb_clauses := nb_clauses + 1;

tmp_var := -2;
formula[i][0] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

tmp_var := 3;
formula[i][1] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

formula[i][IS_CL] := 0;
formula[i][ST_CL] := 2;
formula[i][SZ_CL] := 2;
END_IF;



(* -3 5 -11 -13 0 <=> S2.!S5.C5.C8 *)
IF C5 AND C8 THEN
i := i + 1;
nb_clauses := nb_clauses + 1;

tmp_var := -3;
formula[i][0] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

tmp_var := 5;
formula[i][1] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

formula[i][IS_CL] := 0;
formula[i][ST_CL] := 2;
formula[i][SZ_CL] := 2;

END_IF;

(* -4 6 -11 -12 0 <=> S3.!S6.C5.C7 *)
IF C3 AND C7 THEN
i := i + 1;
nb_clauses := nb_clauses + 1;
```

```
tmp_var := -4;
formula[i][0] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

tmp_var := 6;
formula[i][1] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

formula[i][IS_CL] := 0;
formula[i][ST_CL] := 2;
formula[i][SZ_CL] := 2;

END_IF;

(* -3 -4 0 <=> S2.S3 *)
i := i + 1;
nb_clauses := nb_clauses + 1;

tmp_var := -3;
formula[i][0] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

tmp_var := -4;
formula[i][1] := tmp_var;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0] + 1;
var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
   MAX_NUM_VARIABLE][0]] := i;

formula[i][IS_CL] := 0;
formula[i][ST_CL] := 2;
formula[i][SZ_CL] := 2;

(*clause unitaire pour prendre en compte les contraintes simples *)
FOR j := 0 to nb_outputs DO
    IF F0s[j] = TRUE THEN
        i := i + 1;
        tmp_var := -(j + 1);
        nb_clauses := nb_clauses + 1;

        formula[i][0] := tmp_var;
        var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var
           +MAX_NUM_VARIABLE][0] + 1;
        var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
           MAX_NUM_VARIABLE][0]] := i;

        formula[i][IS_CL] := 0;
        formula[i][ST_CL] := 1;
```

```
        formula[i][SZ_CL] := 1;
    ELSE IF F1s[j] = TRUE THEN
        i := i + 1;
        tmp_var := j + 1;
        formula[i][0] := tmp_var;
        nb_clauses := nb_clauses + 1;

        var_in_clause[tmp_var+MAX_NUM_VARIABLE][0] := var_in_clause[tmp_var
            +MAX_NUM_VARIABLE][0] + 1;
        var_in_clause[tmp_var+MAX_NUM_VARIABLE][var_in_clause[tmp_var+
            MAX_NUM_VARIABLE][0]] := i;

        formula[i][IS_CL] := 0;
        formula[i][ST_CL] := 1;
        formula[i][SZ_CL] := 1;
        END_IF;
    END_IF;
END_FOR;
```

IV-3.3 Fonction principale

```
for i := 1 TO (nb_literals+1) do
    status[i] := UNDEF;
end_for;

for i := 0 TO (MAX_NUM_VARIABLE-1) do
    decisions[i] := 0;
    current_assignment[i] := 0 ;
    to_set[i] := 0;
end_for;

distPrecedent := nb_literals;

WHILE(formula_status <> UNSAT AND distPrecedent <> 1) DO

    for i := 0 TO (nb_literals) BY 1 DO
        IF(status[i] = UNDEF) THEN
            picked := i;
            IF(G[picked] = false) THEN
                picked := -picked;
            END_IF;

            nb_decisions := nb_decisions + 1;
            EXIT;
        END_IF;
    end_for;

    IF(i = nb_literals +1) THEN
        j := 0;
        FOR n := 1 TO nb_literals DO
            IF(status[n] > 0) THEN
                tmp_var := -n;
                mem[j] := true;
            ELSE
                tmp_var := n;
                mem[j] := false;
            END_IF;
```

```
            formula[nb_clauses][j] := tmp_var;

            nb := tmp_var+MAX_NUM_VARIABLE;
            var_in_clause[nb][0] := var_in_clause[nb][0] + 1;
            var_in_clause[nb][var_in_clause[nb][0]] := nb_clauses;

            j := j + 1;
        END_FOR;

        (* HAMMING entre G et mem*)
        dist := 0;
        FOR n := 0 TO (nb_literals-1) DO
            IF G[n] <> mem[n] THEN
                dist := dist + 1;
            END_IF;
        END_FOR;

        IF dist < distPrecedent THEN
            distPrecedent := dist;
            FOR n := 0 TO (nb_literals-1) DO
                A[n] := mem[n];
            END_FOR;
        END_IF;

        nb_clauses := nb_clauses + 1;
        nb_solutions := nb_solutions + 1;
        formula_status := UNDEF;

        IF(dist <> 1) THEN
            backtracking();
        END_IF;

    END_IF;

    IF(distPrecedent <> 1) THEN
        to_set[_to_set] := picked;
        _to_set := _to_set +1;
        decisions[_decisions] := picked;
        _decisions := _decisions + 1;

        WHILE(_to_set > 0 AND formula_status <> UNSAT) DO
            _to_set := _to_set - 1 ;
            var_to_set := to_set[_to_set];

            IF(var_to_set > 0) THEN
                status[abs(var_to_set)] := 1;
            ELSE
                status[abs(var_to_set)] := 0;
            END_IF;

            current_assignment[_current_assignment] := var_to_set;
            _current_assignment := _current_assignment + 1;

            (* 1/ SATisfy clauses containing var_to_set *)
            FOR i := 1 TO var_in_clause[var_to_set+MAX_NUM_VARIABLE][0] DO
                c := var_in_clause[var_to_set+MAX_NUM_VARIABLE][i];
```

```
                formula[c][IS_CL] := formula[c][IS_CL]+1;
            END_FOR;

            (* 2/ Reduce clause containing -var_to_set *)
            FOR i := 1 TO var_in_clause[-var_to_set+MAX_NUM_VARIABLE][0] DO
                c := var_in_clause[-var_to_set+MAX_NUM_VARIABLE][i];
                formula[c][ST_CL] := formula[c][ST_CL]-1;
            END_FOR;

            (* 3/ Verify the clauses *)
            IF(formula_status <> UNSAT) THEN
                FOR i := 1 TO var_in_clause[-var_to_set+MAX_NUM_VARIABLE
                    ][0] DO
                    c := var_in_clause[-var_to_set+MAX_NUM_VARIABLE][i];

                    (* a/ Is a clause Unsatisfied ? *)
                    IF(formula[c][ST_CL] = 0) THEN
                        backtracking();
                    END_IF;

                    (* b/ Is a clause Unit ? *)
                    IF(formula[c][ST_CL] = 1 AND formula[c][IS_CL] = 0)
                        THEN
                        FOR j := 0 TO (formula[c][SZ_CL]-1) DO
                            IF(status[abs(formula[c][j])] = UNDEF) THEN
                                to_set[_to_set] := formula[c][j];
                                _to_set := _to_set + 1 ;
                            END_IF;
                        END_FOR;
                    END_IF;
                END_FOR;
            END_IF;
        END_WHILE;
    END_IF;
END_WHILE;
```

IV-3.4 Back-tracking

```
_current_assignment := _current_assignment - 1;
var_to_unset := current_assignment[_current_assignment];

(* Backtracking *)
WHILE((var_to_unset <> decisions[_decisions-1]) AND (_current_assignment >
    0) AND (formula_status <> UNSAT)) DO
    current_assignment[_current_assignment] := 0;
    status[abs(var_to_unset)] := UNDEF;
    FOR j := 1 TO var_in_clause[-var_to_unset+MAX_NUM_VARIABLE][0] DO
        c := var_in_clause[-var_to_unset+MAX_NUM_VARIABLE][j];
        formula[c][ST_CL] := formula[c][ST_CL] + 1;
    END_FOR;
    FOR j := 1 TO var_in_clause[var_to_unset+MAX_NUM_VARIABLE][0] DO
        c := var_in_clause[var_to_unset+MAX_NUM_VARIABLE][j];
        formula[c][IS_CL] := formula[c][IS_CL] - 1;
    END_FOR;
    _current_assignment := _current_assignment -1;
    var_to_unset := current_assignment[_current_assignment];
END_WHILE;
```

```
status[abs(var_to_unset)] := UNDEF;

FOR j := 1 TO var_in_clause[-var_to_unset+MAX_NUM_VARIABLE][0] DO
    c := var_in_clause[-var_to_unset+MAX_NUM_VARIABLE][j];
    formula[c][ST_CL] := formula[c][ST_CL] + 1;
END_FOR;
FOR j := 1 TO var_in_clause[var_to_unset+MAX_NUM_VARIABLE][0] DO
    c := var_in_clause[var_to_unset+MAX_NUM_VARIABLE][j];
    formula[c][IS_CL] := formula[c][IS_CL] - 1;
END_FOR;
_decisions := _decisions -1;

(* Assign decision to -decision *)
IF(_decisions >= 0) THEN
    decisions[_decisions] := 0;
    current_assignment[_current_assignment] := 0;
    current_assignment[_current_assignment] := -var_to_unset;
    WHILE(_to_set >= 0) DO
        to_set[_to_set] := 0;
        _to_set := _to_set -1 ;
    END_WHILE;
    _to_set := 0;
    to_set[_to_set] := -var_to_unset;
    _to_set := _to_set + 1;

ELSE
    formula_status := UNSAT;
END_IF;
```

Résumé

Cette thèse contribue à une approche formelle de conception d'un programme de contrôle/commande pour les systèmes automatisés de production (SAP) contrôlés par des automates programmables industriels (API). Dans ce contexte, deux constats principaux ont été soulevés : il existe un manque de méthodologie efficace pour la conception d'un programme API dans le monde industriel et les méthodes formelles issues du monde académique ne sont ni connues ni utilisées par l'industrie car trop complexes. Par ailleurs, l'industrie du futur nécessitera des contrôleurs toujours plus flexibles et fiables. La flexibilité implique que les programmes seront encore plus difficiles à réaliser, et par conséquent, la difficulté pour garantir la fiabilité de ceux-ci sera accrue.

Pour répondre à ces problématiques, une méthode de conception formelle s'intégrant dans un cycle de développement industriel classique (cycle en V) a été proposée. De plus, afin de faciliter le transfert vers l'industrie tant d'un point de vue technique (API) qu'humain (pratique des automaticiens), le formalisme utilisé est entièrement basé sur des variables et des équations logiques appelées contraintes logiques. Ces contraintes logiques permettent la spécification des exigences informelles recensées dans le cahier des charges. A partir de ces contraintes logiques, un algorithme de résolution des contraintes, implémentable dans un API, est synthétisé et implémenté automatiquement dans un langage de programmation normalisé pour API. Ce filtre logique peut être utilisé pour : commander un SAP contrôlé par un API, vérifier formellement un programme API, mettre en sécurité un programme API déjà existant présentant des erreurs.

Les travaux de cette thèse ont eu pour objectif de lever certains verrous et de globalement améliorer et renforcer l'approche par filtre logique. Dans le but de généraliser l'approche par filtre, un effort important a été réalisé autour de la formalisation des contraintes logiques et des différentes fonctions et propriétés associées au filtre logique. Cet apport de formalisation a permis, en particulier, de proposer une approche de vérification formelle de la notion de cohérence d'un filtre logique ainsi qu'une condition nécessaire et suffisante à cette propriété. Enfin, après avoir mis à jour l'algorithme d'implémentation classique, deux algorithmes de recherche locale d'une solution basés sur des techniques de solveur SAT ont été proposés.

Commande, sécurité fonctionnelle, systèmes manufacturiers, automate programmable industriel, filtre logique

Abstract

This thesis contributes to a formal approach to design control/command program for automated production systems controlled by Programmable Logical Controller (PLC). In this context, two main observations have been highlighted: there is a lack of efficient methodology for the design of PLC program in the industrial field and the academicals formal approaches are neither known nor used in manufacturing industry due to high complexity. Furthermore, the industry of future will require flexible and reliable PLC program. The flexibility implies that programs will be even more difficult to design and, as a consequence, the complexity to guarantee the reliability will be increased.

To address these issues, a formal design approach, presented as a classical V-cycle, have been proposed. Moreover, to facilitate the industrial transfer from both technical (PLC) and human (engineer practice) point of view, the formalism is exclusively based on logical variables and equations called logical constraints. These constraints are used to specify the informal requirements described in the specification book. From these constraints, a logical filter is synthesized automatically and a solving algorithm, IEC 61131-3 compliant, is implemented in the PLC program. This logical filter may be used to: command an automated production system controlled by a PLC, verify formally a PLC program, make safe an existing PLC program containing errors.

The contributions of this thesis covered the whole development cycle: formal specification, formal analysis and synthesis, automatic implementation in a PLC program. To support these contributions, a significant effort was made on the formalism based on logical constraints. This new formalism has allowed, in particular, to proposed a necessary and sufficient condition to the coherence property of a logical filter and to guarantee the convergence of the online solving algorithm. At least, the classical solving algorithm has been updated according to the new formalism, and two algorithms based on SAT solver techniques and local research have been proposed and tested on real PLC.

Control, safety, manufacturing systems, programmable logical controller, logical filter

Discipline : AUTOMATIQUE, SIGNAL, PRODUCTIQUE, ROBOTIQUE

Spécialité : Automatique et Traitement du Signal

Université de Reims Champagne-Ardenne

CRESTIC - EA 3804

Sciences Exactes et Naturelles, Moulin
de la Housse, 51867 Reims, France