La commande par contraintes logiques de sécurité : principe, applications et mise en œuvre

B. Riera, A. Philippot, D. Annebicque et F. Gellot
CReSTIC, Université de Reims Champagne-Ardenne, France.
bernard.riera@univ-reims.fr, alexandre.philippot@univ-reims.fr,
david.annebicque@univ-reims.fr, francois.gellot@univ-reims.fr

Résumé

Cet article présente pour la première fois dans son ensemble l'approche de commande par contraintes logiques de sécurité développée au CReSTIC depuis plusieurs années. Le principe consiste à ajouter à la fin du programme API (Automates Programmables Industriels), avant la mise à jour des sorties, un code spécifique (appelé « filtre logique ») permettant la détection d'une erreur de commande et sa correction (ou compensation). L'approche revient à séparer les aspects fonctionnels et sécuritaires du contrôleur et à considérer un programme API existant comme intervenant uniquement dans la partie fonctionnelle du contrôleur. L'ensemble des contraintes de sécurité est conçu indépendamment du programme API et dépend seulement de la Partie Opérative étudiée. Trois utilisations possibles du filtre logique sont possibles : bloquant, superviseur et contrôleur, permettant entre autres de garantir la sécurité de programmes API existants de façon non intrusive. Les atouts principaux de l'approche sont : la facilité d'implémentation dans un API et son acceptabilité du fait qu'elle peut s'adapter à des programmes existants sans avoir à fondamentalement modifier les méthodes de travail des automaticiens. Les contraintes de sécurité logiques peuvent être vérifiées formellement hors ligne par « *model-checking* ». Toutefois, en amont de cette étape « lourde » de vérification, il est important de s'assurer simplement de la cohérence du filtre logique. Cet article propose à ce titre quelques propriétés nécessaires pour la cohérence des contraintes de sécurité. Un exemple pédagogique «virtuel» de système manufacturier conçu à partir du logiciel de simulation « FACTORY I/O » de la société Real Games permet d'illustrer la méthode développée.

1 Introduction

Parallèlement à l'accroissement de la complexité des Systèmes Automatisés de Production (SAP) en termes de quantité, de besoins en communication, de diversité des composants, etc., les exigences des utilisateurs concernant la sûreté de fonctionnement et l'aide à la conception de programmes se sont également accrues. Les Automates Programmables Industriels (API) constituent encore la principale architecture d'implantation de la commande des SAP et se programment avec des langages normalisés (IEC 61131-3, 2013). Même s'il existe des méthodes et outils de génération automatique de code API, c'est encore souvent la compétence et l'expérience de l'automaticien qui lui permettent d'élaborer les programmes API. Pouvoir assurer de manière formelle la sécurité des systèmes automatisés est donc aujourd'hui un défi scientifique porteur d'enjeux industriels importants.

Cet article présente dans son ensemble une méthode originale de conception des contrôleurs logiques permettant également, de façon non intrusive, de rendre sûrs de fonctionnement des programmes API existants. Dans ce travail, les contrôleurs de type API avec des Entrées (capteurs) et

des Sorties (actionneurs) logiques (i.e. de type Tout ou Rien (TOR)) sont considérés comme des Systèmes à Evénements Discrets (SED) (Cassandras *et al.*, 1999). Les méthodes de tests ou encore de mise en service « virtuelle » (*virtual commissioning*) permettent de vérifier les spécifications sans toutefois les garantir compte tenu de la non exhaustivité des séquences de test. Cela n'est pas le cas des méthodes formelles (comme le « *model checking* ») mais elles nécessitent une expertise qui n'est pas encore celle de l'automaticien aujourd'hui. En complément, les méthodes algébriques (Hietter *et al.*, 2008) ou de type « Supervisory Control Theory (SCT) » (Ramadge *et al.*, 1989) permettent de synthétiser un contrôleur sûr de fonctionnement mais entrainent une nouvelle façon de voir la conception du contrôleur. Or, il n'est pas évident de faire accepter des changements méthodologiques dans le monde industriel où, à titre d'exemple, le langage LADDER (IEC 61131-3) reste encore le plus utilisé. L'idée proposée est d'ajouter à la fin du programme API, un code spécifique filtrant les commandes erronées. Le principe est de définir des contraintes de sécurité logiques vérifiées formellement, autorisant ou non les ordres envoyés à la Partie Opérative (PO) en vue de garantir la sécurité. L'approche revient à séparer les aspects fonctionnels et sécuritaires du contrôleur (Riera *et al.*, 2014) et à considérer le programme API existant comme intervenant uniquement dans la partie fonctionnelle du contrôleur. L'ensemble des contraintes de sécurité est donc conçu indépendamment du programme API par un expert.

L'approche par filtre logique des erreurs de commande et le formalisme utilisé sont décrits dans la première partie de l'article. Celui-ci peut sembler assez inhabituel mais il permet de prendre en compte le fonctionnement cyclique et séquentiel des API. Les contraintes de sécurité sont modélisées sous la forme d'un monôme logique, impliquant une ou plusieurs variables de sortie, et devant être faux à la fin de chaque cycle de l'API pour garantir la sécurité. La deuxième partie de l'article décrit les 3 possibilités d'utilisation des contraintes logiques : filtre bloquant, superviseur et contrôleur. Dans le premier cas, lorsqu'une contrainte est violée, la Partie Commande (PC) est bloquée dans un état sûr et en empêchant une évolution de la PO. L'approche superviseur consiste à corriger les erreurs de commande sans empêcher par la suite le programme API d'évoluer. En d'autres termes, cela permet dans certains cas de réaliser tout de même les objectifs fonctionnels, même si le programme API existant comporte des erreurs. L'approche « contrôleur » est identique à l'approche « superviseur ». La différence réside dans le fait que la conception de la commande prend en compte l'existence du filtre et permet par là même de séparer les parties fonctionnelles et sécuritaires du programme API. Cette approche modifie la façon de concevoir un programme API mais offre des perspectives intéressantes. Les contraintes de sécurité logiques peuvent être vérifiées formellement hors ligne par « model-checking ». Toutefois, il semble également pertinent, en amont de cette étape « lourde » de vérification, de pouvoir vérifier simplement la cohérence des contraintes de sécurité. La troisième partie de l'article propose 4 propriétés structurelles que l'ensemble des contraintes de sécurité doit nécessairement respecter. L'algorithme de détection et de correction des erreurs de commande du filtre logique fait l'objet de la quatrième partie de l'article. Enfin un exemple pédagogique, « les 4 vérins », est proposé dans la dernière partie de l'article pour illustrer la méthode proposée.

2 Filtre de commande par contraintes de sécurité logiques

Un API est un dispositif électronique programmable destiné à la commande de processus industriels dans lequel le programme est exécuté de façon séquentielle et cyclique (cf. Figure 1). Il envoie des ordres (o) vers les préactionneurs de la PO à partir de données d'entrées (i) (i.e. capteurs), de consignes et d'un programme informatique. Durant un cycle de l'API, les entrées (i) sont lues sur les coupleurs d'entrées, le programme API calcule ensuite l'état des sorties (o_k) à partir des (i) et les met à jour en fin de cycle. Tous les calculs sont effectués à partir d'une mémoire image permettant de garantir que les variables d'Entrées/Sorties (E/S) n'évoluent pas pendant le cycle courant. En d'autres

termes, à chaque cycle, l'API calcule la valeur des variables de sorties (variables « commandables ») à partir des variables d'entrées et de mémoires internes (variables « non commandables »). La mise à jour des sorties se fait donc à partir des dernières valeurs calculées des sorties. La lecture des entrées et l'écriture des sorties sont gérées par l'API et donc transparentes pour l'automaticien. Un contrôleur logique implanté dans un API est donc un programme informatique qui calcule à chaque cycle de l'API les variables de sorties à partir de fonctions logiques des variables non commandables (i.e. en lecture seule).

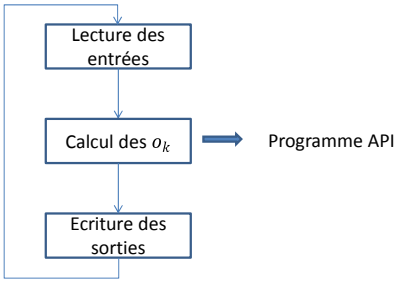


Figure 1: Fonctionnement cyclique et séquentiel de l'API

L'idée proposée est d'intercaler un filtre logique entre le calcul des sorties (qui correspond au programme de l'API) et l'écriture effective de celles-ci sur les coupleurs de sorties en vue de garantir la sécurité de la commande (cf. Figure 2). Le rôle du filtre logique est de traiter les erreurs de commande. Pour cela, il doit assurer 2 fonctions :

1) détecter l'existence d'un état du vecteur de sorties incorrect, il s'agit de la fonction de détection,

2) remplacer l'état incorrect (non sûr) du vecteur de sorties par un état correct (conforme à la sécurité), il s'agit de la fonction de correction (ou de compensation).

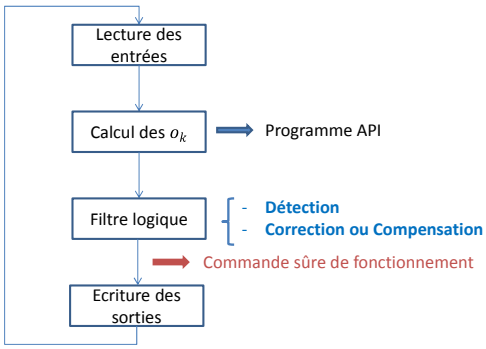


Figure 2: Approche par filtre logique des erreurs de commande

Les notations utilisées dans l'article sont les suivantes :

- t : cycle API courant, *t-1* cycle API précédent.
- $o_k = o_k(t)$: variable logique associée à la valeur de la $k^{ième}$ sortie logique de l'API à t. Les sorties à t sont considérées comme les seules variables pouvant être commandées (variables en lecture/écriture) à chaque cycle API. Toutes les autres variables de l'API (entrées, sorties précédentes, …) sont non commandables et en lecture seule.
- $o_k^* = o_k(t-1)$: variable logique (en lecture seule) associée à la valeur de la $k^{ième}$ sortie logique de l'API à *t-1* (cycle API précédent).
- "$\cdot$, +, $\oplus$, $\bar{x}$" sont respectivement les opérateurs logiques AND, OR, XOR et NOT x.

- 0 signifie FAUX et 1 signifie VRAI.
- $\sum$ et $\prod$ sont respectivement la somme logique et le produit logique de variables logiques.
- $\sum\prod$ est un polynôme logique (somme de produits également appelée forme SIGMA-PI).
- $\uparrow$ x est le front montant de la variable logique x (dans l'API, $\uparrow x = \overline{x^*} \cdot x$).
- $\downarrow$ x est le front descendant de la variable logique x (dans l'API, $\downarrow x = x^* \cdot \bar{x}$).
- O : ensemble des variables de sorties à *t*.
- Y : ensemble des variables non commandables (en lecture seule) à *t, t-1, t-2* …
- CS : ensemble des contraintes de sécurité, N_o : nombre de sorties de l'API.
- N_{CSs}: nombre de contraintes de sécurité simples.
- N_{CSc}: nombre de contraintes de sécurité combinées.
- N_{CFs}: nombre de contraintes fonctionnelles simples.

L'approche proposée pour assurer la sécurité des programmes API est basée sur l'utilisation de contraintes logiques de sécurité qui vont permettre de garantir qu'aucun état dangereux de la PO ne sera jamais atteint. Celles-ci vont agir comme des "gardes", qui placées à la fin du programme API, avant la mise à jour des sorties, vont autoriser ou interdire les ordres de commande envoyés à la PO (Marangé *et al.* 2007). L'ensemble des contraintes de sécurité agit donc comme un filtre logique de commandes erronées.

Les contraintes sont toujours conçues avec le point de vue de l'API (fonctionnement cyclique et séquentiel, changement simultané possible des entrées, ...) car elles ont pour vocation à être implémentées dans celui-ci. Pour cela, l'approche est orientée « signaux » et non « événements ».

Il est considéré que l'état initial est sûr et que toutes les sorties (o_k) sont à 0 et que la PO n'est pas défaillante. Les contraintes doivent toutefois permettre au système de production d'atteindre les objectifs de production pour lequel il a été conçu. Par exemple, un ensemble de contraintes logiques qui met à zéro toutes les sorties à chaque cycle API garantit la sécurité mais nécessairement pas les objectifs de production.

Dans la formalisation proposée, les contraintes de sécurité sont représentées au moyen d'un monôme logique pouvant impliquer à *t* une seule sortie (contraintes de sécurité simples : *CSs*) ou plusieurs sorties (contraintes de sécurité combinées : *CSc*). L'écriture des contraintes nécessite la connaissance des E/S aux cycles courant (*t*) et éventuellement précédent (*t-1*, si utilisation de fronts par exemple). Il peut aussi être nécessaire de définir des variables supplémentaires (observateurs) pour pallier au manque d'observabilité du système. Ces observateurs sont en fait des mémoires de l'état du système et sont non commandables.

L'ensemble des contraintes de sécurité CS est considéré comme nécessaire et suffisant pour garantir la sécurité. En d'autres termes, ajouter une contrainte supplémentaire est sans effet sur la sécurité. En revanche, en retirer une ne permet plus de la garantir.

Les *CSs* et *CSc* sont représentées (respectivement équations (1) et (2)) sous la forme d'un monôme logique (Π, produit de variables logiques) impliquant une seule variable de sortie : o_k (*CSs*) ou plusieurs variables de sortie : o_k, o_l … (*CSc*) au cycle API courant *t* et des variables non commandables (Y).

$$\forall m \in [1, N_{CSs}],\ \exists! k \in [1, N_o] \ /\ \ CSs_m = \prod(o_k, Y) \quad (1)$$

$$\forall n \in [1, N_{CSc}],\ \exists! (k, l, \dots) \in [1, N_o]\ avec\ \ k \neq l \neq \cdots \ /\quad CSc_n = \prod(o_k, o_l, \dots, Y) \quad (2)$$

Les *CSs* et *CSc* doivent être nécessairement toutes FAUSSES (=0) avant la mise à jour des sorties par l'API pour garantir la sécurité. Par conséquent, la somme logique de toutes les contraintes de sécurité calculée avec les valeurs de sorties o_k doit être fausse avant la mise à jour des sorties sur les coupleurs de l'API pour garantir la sécurité (équation 3). Il s'agit de la fonction détection du filtre logique. Un programme API est donc considéré comme sûr si aucune contrainte de

sécurité n'est violée avant la mise à jour des sorties. Le vecteur de sorties $(o_1, \dots, o_k, \dots, o_{No})$ est considéré comme sûr si l'équation (3) est vérifiée en fin de cycle de l'API.

$$\sum_{i=1}^{N_{CSs}} CSs_i + \sum_{\mathrm{j}=1}^{N_{CSc}} CSc_{\mathrm{j}} = 0 \quad (3)$$

L'écriture des contraintes sous la forme de monômes logiques présente plusieurs avantages. D'une part, elle est simple à comprendre pour l'automaticien et est adaptée à sa vision « système » et « sous-systèmes ». D'autre part, les contraintes sont très faciles à implémenter dans un API. La séparation des contraintes de sécurité en *CSs* et *CSc* est également intéressante. En effet, lorsqu'une *CSs* est violée, il n'y a nécessairement qu'une seule solution pour la soulager : complémenter la sortie impliquée dans la contrainte. En revanche, soulager une *CSc* nécessite un choix de l'expert et ne doit pas conduire à violer des *CSs* et d'autres *CSc*. Il existe seulement 2 formes exclusives possibles pour les contraintes de sécurité simples (*CSs*) car elles sont exprimées sous la forme d'un monôme logique impliquant une seule sortie à *t*. Par conséquent, les *CSs* peuvent s'écrire de la façon suivante :

$$\forall m \in [1, \mathrm{N_{CSs}}],\ \exists!\, k \in [1, \mathrm{N_O}] \,/\ \ CSs_m = o_k \cdot h_{0m}(\mathrm{Y}) + \overline{o_k} \cdot h_{1m}(\mathrm{Y})$$
$$\text{avec } h_{0m}(\mathrm{Y}) \oplus h_{1m}(\mathrm{Y}) = 1 \quad (4)$$

Les contraintes de sécurité simples (*CSs*) expriment le fait que pour avoir la contrainte égale à 0 à *t* et donc garantir la sécurité en fin de cycle de l'API avant la mise à jour des sorties, si le monôme $h_{0m}(\mathrm{Y})$ est VRAI, o_k doit être nécessairement FAUX. En revanche, si le monôme $h_{1m}(\mathrm{Y})$ est VRAI, o_k doit être nécessairement VRAI. La somme logique de toutes les *CSs* (qui doit être également fausse) peut s'écrire selon l'équation (5).

$$\sum_{i=1}^{N_{CSs}} CSs_i = \sum_{k=1}^{N_o} \left(f_{sk}(o_k, Y) \right) \quad (5)$$

Où $f_{sk}(o_k, \mathrm{Y})$ est un polynôme logique (forme SIGMA-PI), indépendant des autres sorties à *t* car seulement les *CSs* sont considérées. $f_{sk}(o_k, \mathrm{Y})$ peut être développée (équation (6)) où f_{s0k} et f_{s1k} sont des polynômes logiques (forme SIGMA-PI) impliquant uniquement des variables non commandables. Pour simplifier l'écriture des équations, une fonction logique est également représentée par une variable logique de même nom.

$$f_{sk}(o_k, Y) = o_k \cdot f_{s0k}(Y) + \overline{o_k} \cdot f_{s1k}(Y) = o_k \cdot f_{s0k} + \overline{o_k} \cdot f_{s1k} \quad (6)$$

A partir des équations (5) et (6), il est possible d'écrite l'équation (7).

$$\sum_{i=1}^{N_{CSs}} CSs_i = \sum_{k=1}^{N_o} \left(o_k \cdot f_{s0k}(\mathrm{Y}) + \overline{o_k} \cdot f_{s1k}(\mathrm{Y}) \right) = \sum_{k=1}^{N_o} (o_k \cdot f_{s0k} + \overline{o_k} \cdot f_{s1k}) \quad (7)$$

De la même manière que pour les contraintes de sécurité simples, il est possible d'introduire les fonctions f_{c0k} et f_{c1k} pour indiquer pour chaque contrainte de sécurité combinée *CSc*, si la sortie o_k doit être forcée à 0 ou à 1 lorsqu'elle est violée.

$$f_{c0k} = f_{c0k}(\mathrm{o_k}, \mathrm{o_l}, \dots, \mathrm{Y}) = \sum_{\mathrm{j}=1}^{\mathrm{N_{CSc}}} \mathrm{MC0}_{k,j} \cdot CSc_j \quad (8)$$

$$f_{c1k} = f_{c1k}(\mathrm{o_k}, \mathrm{o_l}, \dots, \mathrm{Y}) = \sum_{\mathrm{j}=1}^{\mathrm{N_{CSc}}} \mathrm{MC1}_{k,j} \cdot CSc_j \quad (9)$$

$\mathrm{MC0}_{k,j}$ et $\mathrm{MC1}_{k,j}$ sont des variables booléennes dont les valeurs sont définies par l'expert lors de la phase de définition des contraintes de sécurité combinées. Elles indiquent respectivement si la sortie o_k doit être forcée à 0 ou à 1 lorsque la contrainte de sécurité combinée *j* est violée.

$$\forall k \in [1, \mathrm{N_O}], \forall j \in [1, \mathrm{N_{CSc}}],\ \ MC0_{k,j} = 1 \ si\ o_k\ doit\ \hat{e}tre\ forc\acute{e}e \text{ à 0 quand } CSc_j = 1$$
$$sinon\ MC0_{k,j} = 0$$
$$\forall k \in [1, \mathrm{N_O}], \forall j \in [1, \mathrm{N_{CSc}}],\ \ MC1_{k,j} = 1 \ si\ o_k\ doit\ \hat{e}tre\ forc\acute{e}e \text{ à 1 quand } CSc_j = 1$$
$$sinon\ MC1_{k,j} = 0 \quad (10)$$

Il est possible de représenter les équations (8), (9) et (10) sous forme matricielle en considérant *CSc*, f_{c0} et f_{c1} des vecteurs colonnes et MC0 et MC1 des matrices de dimension (N_o, N_{CSc}).

Les premiers travaux de recherche du CReSTIC sur la commande par contraintes (Marange *et al.*, 2007) (Benlorfhar *et al*, 2011) ont conduit à proposer une méthode formelle de « *model-checking* » (Behrmann *et al*., 2002) pour vérifier hors ligne que l'ensemble de contraintes CS, quel que soit le contrôleur logique, permet d'éviter que la PO atteigne un état dangereux (fonction détection du filtre) et que le système reste commandable. En d'autres termes, qu'il existe bien une séquence de sorties permettant d'atteindre les objectifs de production. Le problème de la compensation de la commande (fonction correction du filtre logique) avait alors été simplifiée car il était considéré qu'il existait un vecteur de sorties connu permettant toujours de mettre la PO dans un état de repli sûr. Cette hypothèse est toutefois restrictive. Si l'on considère qu'il n'existe pas de vecteur de sorties prédéfini capable de sécuriser la commande lorsqu'une contrainte est violée, il est nécessaire de déterminer un **vecteur de sorties cohérent** avec l'ensemble des contraintes. **Cela signifie que lorsqu'une contrainte est violée, il doit exister au moins un vecteur de sorties tel qu'aucune contrainte ne soit violée.** Par conséquent, il semble pertinent en amont de l'étape difficile de « model-checking », de pouvoir vérifier simplement la cohérence des contraintes de sécurité. Cet article propose à ce titre quelques propriétés structurelles nécessaires pour la cohérence des contraintes de sécurité. Ce point sera étudié après une présentation des différentes utilisations possibles du filtre logique.

3 Applications du filtre logique

Nous proposons 3 applications du filtre logique dans un API : le filtre logique « bloquant », « superviseur » ou « contrôleur ». Le principe reste le même dans les 3 cas, le filtre logique est un ajout de code dans l'API juste avant la mise à jour des sorties. La différence principale réside dans la prise en compte ou non du filtre logique dans la phase de conception du contrôleur. Dans le cas du filtre « bloquant » et du filtre « superviseur », on considère que le programme API existe et qu'il a été conçu indépendamment du filtre logique. Par conséquent, le programme API, s'il est bien fait, dans ces 2 cas d'utilisation, ne devrait jamais déclencher le filtre logique.

3.1 Filtre logique "bloquant"

L'utilisation en tant que filtre bloquant consiste, lorsqu'une contrainte de sécurité est violée, à replier la PO dans un état sûr en stoppant l'évolution de la commande (cf. Figure 3). Contrairement à l'approche initiale (Marange *et al.*, 2007) (Benlorfhar *et al*, 2011) où un vecteur de sorties supposé connu permettait toujours de mettre la PO dans un état de repli sûr, une solution générique est proposée. Pour cela, si lors du cycle courant ou lors d'un des cycles précédents de l'API, la commande conçue et implémentée par l'automaticien viole ou a violé une contrainte, on cherche le vecteur de sorties sûres en repliant par défaut toutes les sorties. La fonction « correction » consiste donc à trouver un vecteur de sorties sûres en considérant qu'aucune commande est envoyée à la PO. Ainsi, la PO se stabilisera nécessairement dans un état de repos sûr proche ou similaire à l'état initial. Cette approche présente un intérêt pour la télésurveillance des installations industrielles mais également pour la formation pratique des automaticiens. Il est important que les apprenants puissent mettre en application sur des Parties Opératives (réelles ou simulées) les concepts théoriques de l'automatique et acquérir des savoir-faire en adéquation avec les réalités et les besoins du monde industriel (Riera *et al.*, 2009). Même si l'utilisation de la simulation est une solution séduisante, le fait

de travailler sur un système réel reste incontournable mais soulève plusieurs inconvénients liés à la sécurité des hommes et des biens. L'approche par filtre bloquant permet de fiabiliser et de stopper la PO suite à une erreur de commande de l'apprenant. On notera également que les contraintes peuvent dans ce cadre être utilisées comme une source d'explications pour les apprenants (Marangé *et al.*, 2014).

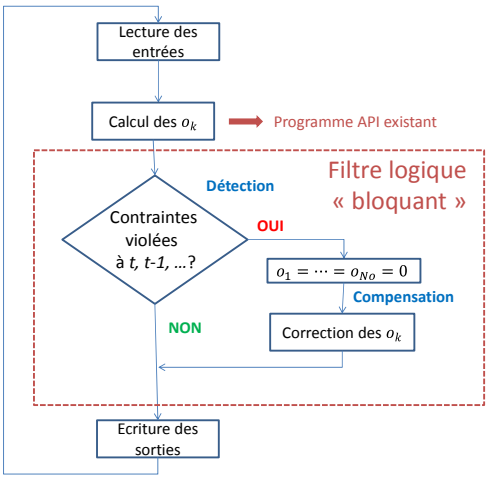


Figure 3 : Principe du filtre logique « bloquant »

3.2 Filtre logique "superviseur"

À la fin des années 80, une contribution majeure dans le domaine des systèmes à évènements discrets a été proposée dans le cadre de la théorie de supervision (SCT) développée par P.J.G. Ramadge et W.M. Wonham (1989). Cette théorie a pour objectif de synthétiser un « superviseur » qui, une fois couplé à un système, impose à celui-ci de respecter des spécifications données. Le système est considéré comme un générateur spontané d'évènements répartis en deux classes : les évènements commandables et non commandables. Pour contraindre le comportement du système à respecter un comportement spécifié, le superviseur peut interdire la génération d'évènements commandables. Pour cela, il dispose de la connaissance de tous les évènements qui ont déjà été générés par le système. Le superviseur permet donc de restreindre le comportement d'un système pour que celui-ci respecte une spécification donnée. Même si les outils et formalismes utilisés dans la SCT (langages, automates à état, événements) sont différents de l'approche par contraintes proposée, les concepts présentent des similarités. Les contraintes logiques peuvent être utilisées en vue de réaliser un « superviseur » de la commande implantée (ou existante) qui peut être erronée. Contrairement au filtre bloquant, le principe consiste alors seulement à modifier les ordres envoyés à la PO si une contrainte de sécurité est violée (cf. Figure 4). Par conséquent, le programme de commande évolue librement mais les ordres sont effectivement envoyés à la PO seulement si les contraintes de sécurité ne sont pas violées. Dans le cas contraire, la fonction de compensation permet d'envoyer des ordres sûrs compatibles avec toutes les contraintes de sécurité. Cette solution permet de sécuriser des programmes API existants tout en continuant éventuellement d'atteindre les objectifs de production. Elle a été testée avec succès dans le cadre de la commande des équipements d'alimentation des lignes électrifiées (EALE) de la SNCF (Coupat *et al.*, 2014), permettant ainsi de fiabiliser une commande existante sans modifier le programme API existant. En effet, les méthodes « académiques » pour synthétiser un contrôleur sûr comme les méthodes algébriques (Hietter, 2008), et la SCT (Ramadge *et al.*, 1989) nécessitent une expertise qui n'est pas encore celle de l'automaticien aujourd'hui. De plus, ces méthodes sont souvent orientées « événements » et non « signaux » et posent donc des difficultés d'implémentation dans les

API (Fabian *et al.*, 1998). L'idée proposée ici est de ne pas modifier les méthodes de travail de l'automaticien mais d'ajouter à la fin du programme existant un code API spécifique permettant de garantir la sécurité et éventuellement de continuer d'assurer les spécifications fonctionnelles. Le filtre logique est conçu indépendamment du programme API par un expert et repose exclusivement sur une analyse de la PO.

3.3 Filtre logique "contrôleur"

Le filtre "contrôleur" est identique à l'approche "superviseur". Il diffère seulement dans le fait que les contraintes de sécurité sont considérées comme une partie intégrante du programme de commande. Contrairement à l'approche « superviseur », le fait de violer des contraintes est donc considéré comme « normal » dans la conception du contrôleur (cf. Figure 4). L'approche revient ainsi à séparer les aspects fonctionnels et sécuritaires du contrôleur (Riera *et al.*, 2014). Les possibilités offertes par cette vision de la commande sont nombreuses. La gestion des synchronisations et des modes de marche et arrêt semble beaucoup plus aisée comme le montrera l'exemple applicatif développé à la fin de l'article. En revanche, cette approche modifie la façon « classique » de concevoir la commande d'un système manufacturier piloté au moyen d'un API.

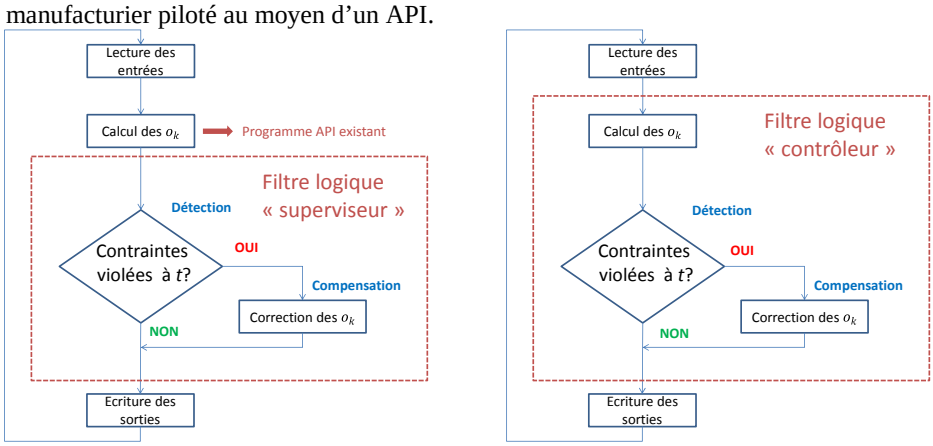


Figure 4 : Principe du filtre logique « superviseur » et « contrôleur »

Même si l'utilisation de la commande par contraintes logiques semble séduisante, elle soulève 2 difficultés majeures : 1) la définition des contraintes 2) la proposition d'un algorithme pour la fonction de correction proposant, lorsque des contraintes sont violées, un vecteur de sorties qui soulage l'ensemble des contraintes de sécurité. Il est donc important de s'assurer qu'un tel vecteur de sorties existe. Si ce n'est pas le cas, il est inutile de réaliser l'étape de vérification des contraintes de sécurité par « model-checking ». Le paragraphe suivant propose des propriétés structurelles que les contraintes de sécurité doivent nécessairement respecter pour éviter un problème d'incohérence de l'ensemble des contraintes.

4 Quelques propriétés structurelles des contraintes de sécurité

La définition des contraintes est la principale difficulté à résoudre pour garantir l'utilisabilité de la méthode. Dans cet article, nous proposons quelques propriétés structurelles nécessaires à la cohérence des contraintes de sécurité. En effet, les contraintes simples et combinées peuvent être antagonistes et dans ce cas l'ensemble des contraintes ne sera pas cohérent. Les contraintes de sécurité simples et

combinées doivent nécessairement respecter la propriété suivante :

Propriété 1 : une condition nécessaire pour garantir la cohérence de l'ensemble des contraintes de sécurité est : $\forall o_k,\ f_{s0k} \cdot f_{s1k} = 0$ et $f_{c0k} \cdot f_{c1k} = 0$ (11)

En effet, si ce n'est pas le cas, cela signifie que 2 contraintes simples ou combinées sont en contradiction, et qu'une des 2 sera nécessairement violée. Par conséquent, l'ensemble des contraintes ne sera pas cohérent. Il est possible de démontrer la proposition suivante :

Proposition : si toutes les contraintes de sécurité simples impliquant la sortie o_k sont écrites exclusivement avec des fronts montants et descendants de la sortie o_k, la propriété 1 est respectée.

Preuve : si toutes les *CSs* impliquant o_k, sont écrites à partir de fronts, en utilisant le théorème d'expansion de Shannon, on peut écrire l'équation (12).

$\forall o_k,\ f_{s0k} = \overline{o_k{}^*} \cdot f_{s0k}$ et $f_{s1k} = o_k{}^* \cdot f_{s1k}$ (12)

Par conséquent, comme $\overline{o_k{}^*} \cdot o_k{}^* = 0$, et que l'état initial est supposé sûr, la propriété 1 est respectée.

$\forall o_k,\ f_{s0k} \cdot f_{s1k} = \overline{o_k{}^*} \cdot f_{s0k} \cdot o_k{}^* \cdot f_{s1k} = 0$ (13)

Dans le cas des *CSs*, lorsqu'une contrainte est violée, la seule solution pour la soulager est de complémenter la valeur de la sortie o_k impliquée dans la contrainte. Dans le cas des contraintes de sécurité combinées (*CSc*), il est nécessaire de choisir une solution particulière en forçant une ou plusieurs des sorties impliquées dans la contrainte combinée dans le respect des autres contraintes de sécurité. Nous proposons une représentation sous la forme d'un graphe de contraintes combinées proche de celle utilisée dans les réseaux d'automates booléens (Tournier, 2005) en vue d'extraire des propriétés structurelles que les contraintes de sécurité combinées doivent respecter. Ce graphe G (G = (V, E)) comprend N_{CSc} sommets (noté V). L'ensemble des arêtes (noté E) indique que le produit de 2 *CSc* est logiquement nul. En d'autres termes, 2 *CSc* sont adjacentes si au moins une variable de sortie apparait pour l'une sous sa forme complémentée et pour l'autre sous sa forme normale (*i.e.* non complémentée). Par conséquent, si 2 *CSc* sont adjacentes, le fait d'en soulager 1 peut conduire à violer l'autre et réciproquement. Il s'agit d'un graphe non orienté. A titre d'exemple, la Figure 5 propose un ensemble de 5 contraintes combinées, le graphe correspondant et sa matrice d'adjacence. Les variables *i* correspondent à des variables d'entrées (capteurs, variables non commandables) et les *o* sont des sorties (variables commandables). Le graphe représente les liens logiques existants entre les *CSc*. Par exemple le fait que, compte tenu de l'existence des 2 formes (complémentée et non complémentée) de la variable o_1 dans les 2 contraintes *CSc1* et *CSc2*, soulager l'une peut conduire à violer l'autre. Nous adoptons la représentation matricielle suivante en vue d'obtenir la matrice d'adjacence du graphe G. MX0 et MX1 sont 2 matrices de dimension (N_o, N_{CSc}).

$$MX0 = \left(MX0_{i,j}\right) 1 \le i \le N_o, \quad 1 \le j \le N_{CSc}\ avec\ MX0_{i,j} = 1\ si\ \forall Y, \quad o_i \cdot CSc_j = 0, 0\ sinon$$
$$MX1 = \left(MX1_{i,j}\right) 1 \le i \le N_o,\ 1 \le j \le N_{CSc}\ avec\ MX1_{i,j} = 1\ si\ \forall Y,\ \overline{o_i} \cdot CSc_j = 0, 0\ sinon \quad (14)$$

La matrice d'adjacence MX du graphe s'obtient alors à partir d'un produit matriciel utilisant la transposée des matrices MX0 et MX1 permettant ainsi de mettre en évidence les liens entre les variables complémentées ou non des sorties.

$$MX = MX1^T \cdot MX0 + MX0^T \cdot MX1 \quad (15)$$

Comme cela a été précédemment vu, lorsqu'une contrainte combinée est violée, en vue de la soulager, il est nécessaire de choisir une solution en forçant une ou plusieurs sorties à l'état 0 ou 1. Cette opération peut se représenter sous la forme d'un Graphe Orienté (GO). La matrice carrée d'adjacence MC du graphe orienté s'obtient à partir des matrices MC0 et MC1.

$$MC = MC0^T \cdot MX0 + MC1^T \cdot MX1 \quad (16)$$

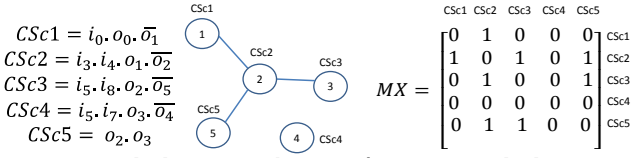


Figure 5 : Exemple de *CSc*, graphe associé G et matrice d'adjacence MX

En reprenant l'exemple précédent, et en supposant que o_1 est prioritaire si *CSc1* est violée, o_2 pour *CSc2*, o_5 pour *CSc3*, o_3 pour *CSc4* et que les 2 sorties o_2 et o_3 doivent mises à 0 si *CSc5*=1, on obtient le graphe orienté et la matrice d'adjacence MC de la Figure 6. L'analyse structurelle du graphe GO permet de faire apparaitre l'existence ou non de cycles. Leur présence signifie que l'ensemble des *CSc*, avec les priorités entre sorties choisies pour les soulager, n'est pas correct car il est alors impossible de trouver un vecteur de sorties qui respecte toutes les contraintes.

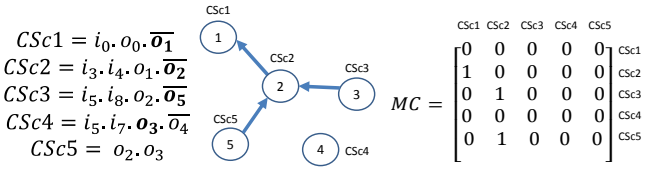


Figure 6 : Exemple de *CSc*, graphe orienté associé GO et matrice d'adjacence MC

Propriété 2 : une condition nécessaire pour la cohérence de l'ensemble des contraintes de sécurité est que le graphe orienté GO des *CSc* ne doit pas présenter de cycles.

De plus, les fc_{0k} et fc_{1k} doivent être cohérents avec les fs_{0k} et fs_{1k}. En d'autres termes, le fait de soulager une contrainte combinée ne doit pas conduire à violer une contrainte simple. Cette condition peut s'exprimer au moyen des 2 propriétés suivantes :

Propriété 3 : une condition nécessaire pour la cohérence de l'ensemble des contraintes de sécurité est : $\forall o_k,\ f_{c0k} \cdot f_{s1k} = 0$ (17)

Propriété 4 : une condition nécessaire pour la cohérence de l'ensemble des contraintes de sécurité est : $\forall o_k,\ f_{c1k} \cdot f_{s0k} = 0$ (18)

On notera toutefois que les fs_{0k} et fs_{1k} sont prioritaires par rapport aux fc_{0k} et fc_{1k} car ces fonctions ne dépendent que de variables non commandables. Les 4 propriétés qui viennent d'être présentées sont nécessaires pour que l'ensemble des contraintes soit cohérent. Elles peuvent être vérifiées facilement avant l'étape de « model-checking ». Le paragraphe suivant propose un algorithme intégrant les 2 fonctions (détection et correction) du filtre logique.

5 Algorithme de détection et de correction de la commande pour le filtre logique

De façon similaire aux contraintes de sécurité (*CS*), nous pouvons définir des contraintes "fonctionnelles" (*CF*) pour indiquer si la sortie o_k doit être égale à 0 ou à 1 d'un point de vue fonctionnel. Comme pour les contraintes de sécurité, les contraintes fonctionnelles doivent être toutes fausses avant la mise à jour des sorties par l'API. Compte tenu du fonctionnement séquentiel de l'API et de la position du filtre logique, seules les contraintes fonctionnelles simples (*CFs*) ont un sens. De plus, habituellement l'automaticien spécifie un contrôleur en indiquant uniquement quand la sortie doit être activée. Par conséquent, contrairement aux *CSs*, nous utiliserons uniquement la forme de *CFs* conduisant à activer une sortie. Les *CSs* s'écrivent donc de la façon suivante où $l_{1m}(Y)$ est un monôme logique faisant intervenir uniquement des variables en lecture seule. Les contraintes

fonctionnelles simples (*CFs*) expriment le fait que pour avoir la contrainte fausse à *t*, si le monôme $l_{1m}(\mathrm{Y})$ est VRAI, alors la sortie impliquée dans la contrainte fonctionnelle doit être activée :

$$\forall m \in [1, \mathrm{N_{CFs}}],\ \exists!\, k \in [1, \mathrm{N_O}]\ /\quad CFs_m = \overline{o_k} \cdot l_{1m}(\mathrm{Y}) = \overline{o_k} \cdot l_{1m} \tag{19}$$

La somme logique de toutes les *CFs* (qui doit être également fausse) peut s'écrire selon l'équation (20).

$$\sum_{i=1}^{N_{CFs}} CFs_i = \sum_{k=1}^{N_o} \left(g_{1k}(o_k, Y)\right) = \sum_{k=1}^{N_o} \left(\overline{o_k} \cdot g_{1k}(Y)\right) = \sum_{k=1}^{N_o} (\overline{o_k} \cdot g_{1k}) \tag{20}$$

Où $g_{1k}(o_k, \mathrm{Y})$ est un polynôme logique (forme SIGMA-PI), indépendant des autres sorties à *t* car seulement les *CFs* sont considérées. Dans le cas du filtre logique « bloquant » ou « superviseur », à chaque cycle API, les g_{1k}correspondent aux sorties o_k obtenues à partir du programme API existant.

Hietter (2008) dans son travail sur la synthèse algébrique a démontré le théorème suivant :

Théorème 1 : Soit *B* une algèbre de Boole. Soient *a*, *b*, *x* trois éléments de cette algèbre. Considérons l'équation suivante où *x* est l'inconnue : $a \cdot \bar{x} + b \cdot x = 0$

Cette équation admet des solutions si et seulement si : $a \cdot b = 0$

Lorsque cette condition est respectée, cette équation admet une ou plusieurs solutions que la forme paramétrique suivante permet de décrire : $x = a + \bar{b} \cdot p$ où p est un paramètre, élément de *B*. Ce théorème peut être appliqué pour trouver une commande permettant de soulager les *CSs* violées. En effet, pour chaque sortie o_k, l'équation : $o_k \cdot f_{s0k} + \overline{o_k} \cdot f_{s1k} = 0$ doit être respectée (cf. équation (7)).

Une solution de cette équation est donc $o_k = \overline{f_{sok}} \cdot p + f_{\mathrm{s1}k}$ ou p est un paramètre booléen. On constate de plus que la propriété 1 proposée dans le paragraphe précédent doit être respectée. Ce théorème permet donc d'apporter une solution à la résolution des contraintes de sécurité simples lorsqu'elles sont violées en intégrant les contraintes « fonctionnelles » qui peuvent être assimilées au paramètre *p*. La solution de l'équation devient l'équation (21).

$$o_k = \overline{f_{sok}} \cdot g_{1k} + f_{\mathrm{s1}k} \tag{21}$$

Le principe est le suivant. Si les contraintes fonctionnelles tentent d'activer une sortie à 1, celle-ci sera effective si les contraintes de sécurité ne la forcent pas à 0. De la même façon, si les contraintes fonctionnelles désactivent une sortie, celle-ci sera effective seulement si les contraintes de sécurité ne la forcent pas à 1. Si aucune *CSs* n'est violée, alors $o_k = g_{1k}$. L'intégration des contraintes de sécurité combinées peut se faire de la même manière en notant que les *CSc* sont prioritaires par rapport aux g_{1k} mais pas vis-à-vis des *CSs*.

$$o_k = \overline{f_{sok}} \cdot \left(\overline{f_{c0k}} \cdot g_{1k} + f_{c1k}\right) + f_{s1k} \tag{22}$$

f_{c0k} and f_{c1k} forcent respectivement la sortie o_k à 0 ou à 1 si des *CSc* sont violées. Toutefois, ce calcul ne garantit pas que la commande o_k ne viole pas d'autres contraintes de sécurité.

Cela nous conduit à proposer l'algorithme de détection et d'accommodation décrit figure 7 pour le filtre logique. A partir du vecteur de sorties g_1, des vecteurs fs_0, fs_1 et des matrices MC0 et MC1, l'algorithme renvoie un vecteur de sorties sûres. Si aucune contrainte de sécurité n'est violée, o_k est égal à g_{1k}. Le principe de l'algorithme (simple à implémenter en langage ST dans un API) consiste à calculer un vecteur de sorties respectant les contraintes de sécurité. Si les sorties calculées violent des contraintes combinées, le calcul est réitéré (boucle WHILE) après avoir calculé les f_{c0k} et f_{c1k} pour soulager les *CSc*. Si les contraintes de sécurité sont mal définies (propriétés structurelles non respectées par exemple) ou si une défaillance de la PO apparait (panne capteur par exemple), aucun vecteur de sorties sûres ne peut être trouvé. Pour éviter une boucle sans fin et le déclenchement du chien de garde de l'API, la boucle WHILE est arrêtée quand le nombre d'itérations est supérieur à $\mathrm{N_{CSc}}$. C'est une solution simple pour déterminer la présence de cycles dans le graphe orienté de

contraintes GO. Si cela, se produit, la commande envoyée consiste à replier les sorties sans tenir compte des contraintes combinées.

```
(* fso, fs1, MC0, MC1, g1 sont connus *)
No : Nombre de sorties, Ncsc : Nombre de contraintes combinées
Pour k de 1 à No faire
                fc0(k) :=FAUX
                fc1(k) :=FAUX
Fin Pour
Flag := VRAI
Cpt :=0
Tant que (Flag and Cpt<Ncsc+1) faire
        Pour k de 1 à No faire
                Sl(k) := not fs0(k) and (not fc0(k) and g1(k) or fc1(k)) or fs1(k)
        Fin Pour
        Flag := FAUX
        Pour i de 1 à Ncsc faire
                Flag := Flag or CSc(i)   [calculer avec les valeurs intermédiaires des sorties : Sl(1), …, Sl(No)]
        Fin Pour
        Cpt := Cpt+1
        Pour k de 1 à No faire
                fc0(k) :=FAUX
                fc1(k) :=FAUX
                Pour i de 1 à Ncsc faire
                        fc0(k) := fc0(k) or (MC0(k, i) and CSc(i)) (. => produit logique matriciel)
                        fc1(k) := fc1(k) or (MC1(k, i) and CSc(i)) (. => produit logique matriciel)
                Fin Pour
        Fin Pour
Fin Tant que
Si cpt>Ncsc Alors (=> ensemble de contraintes de sécurité incohérent)
        Pour k de 1 à No faire
                Sl(k):= not fs0(k) and fs1(k)
        Fin Pour
Fin Si
Pour k de 1 à No faire
        o(k):= Sl(k)
Fin Pour
```

Figure 7 : Algorithme de détection et de correction de la commande du filtre logique

La dernière partie de l'article propose un exemple pédagogique d'utilisation du filtre logique.

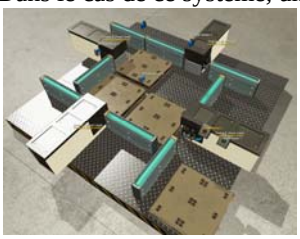
6 Exemple illustratif : les « 4 vérins »

Le logiciel FACTORY I/O de la société Real Games a été utilisé pour concevoir un système industriel simple mais présentant des problèmes de sécurité intéressants. En effet, il est possible avec FACTORY I/O de construire en 3D toutes sortes de PO liées à l'industrie manufacturière et de les connecter à des API matériels ou logiciels. C'est un outil logiciel très intéressant pour la formation des automaticiens et la recherche en systèmes à événements discrets (SED). Une version d'évaluation de FACTORY I/O est disponible sur le site www.realgames.pt.

6.1 Description du système

Le système construit est purement pédagogique. Il s'agit de 4 vérins « simple effet » pouvant faire tourner 1, 2 ou 3 palettes dans le sens horaire en évitant les collisions entre palettes et entre palette et tige de vérin (cf. Figure 8). On peut constater que s'il n'y a pas de palettes, il n'y a aucun problème de

sécurité. En revanche, si 4 palettes sont présentes, le système ne doit accepter aucun mouvement des vérins. Ce système est composé de 4 actionneurs (o1, o2, o3 et o4) pour sortir respectivement les vérins 1, 2, 3 et 4. 12 capteurs indiquent si les vérins sont rentrés (b1, b2, b3, b4) ou sortis (h1, h2, h3, h4) et si des palettes sont présentes devant les vérins (i1, i2, i3, i4). 2 boutons poussoirs (start et stop) viennent compléter l'instrumentation. Différents ensembles de contraintes de sécurité sont possibles. Ils peuvent plus ou moins contraindre la commande. Autoriser le maximum de liberté à la commande (*i.e.* être permissif) nécessite souvent de définir des observateurs pour compléter les capteurs disponibles. Dans le cas de ce système, une solution permissive est proposée.



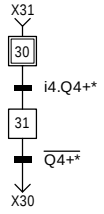
ELEMENTS	Capteurs	Actionneurs
Vérin 1	b_1,h_1	$o_1 = Q_1^+$
Vérin 2	b_2, h_2	$o_2 = Q_2^+$
Vérin 3	b_3, h_3	$o_3 = Q_3^+$
Vérin 4	b_4, h_4	$o_4 = Q_4^+$
Positions	i_1, i_2, i_3, i_4	

Figure 8 : Le système des 4 vérins et E/S correspondantes

6.2 Ensemble de contraintes de sécurité

Le filtre logique repose sur 24 *CSs* (6 par vérin), 4 *CSc* et 4 observateurs (1 par vérin). Cet ensemble de contraintes laisse beaucoup de liberté à la commande. Les 6 contraintes de sécurité simples relatives au vérin 1 et les 4 contraintes de sécurité combinées sont présentées et expliquées (cf. Figure 9).

CSs1, CSs2, CSs3, CSs4 et CSs5 indiquent respectivement qu'il n'est pas autorisé de donner l'ordre de sortie du vérin 1 si a) celui-ci n'est pas rentré, b) si des palettes sont présentes devant les vérins 1 et 2, c) si le vérin 2 n'est pas rentré, d) si une palette est en train d'être transportée par le vérin 4 vers le vérin 1, e) si une palette est devant le vérin 1 et que le vérin 2 est en train de sortir. On notera qu'un observateur est utilisé (étape 31 du GRAFCET active) pour CSs4 en vue de mémoriser si une palette est en cours d'acheminement vers le vérin 1. CSs6 impose de ne pas arrêter l'ordre de sortie du vérin 1 tant que celui-ci n'est pas complétement sorti. Ces contraintes tiennent compte du fonctionnement cyclique et synchrone de l'API ainsi que du retard de causalité. Par analogie, le lecteur pourra écrire les 6 contraintes de sécurité simples spécifiques aux 3 autres vérins. Les 4 contraintes combinées (CSc1, CSc2, CSc3 et CSc4) permettent d'éviter l'envoi d'ordres de sortie simultanés entre 2 vérins adjacents. Toutes les commandes satisfaisant l'ensemble de ces contraintes sont sûres. Ces 28 contraintes de sécurité respectent les 4 propriétés structurelles présentées dans l'article. Cet ensemble de contraintes a également été vérifié formellement hors ligne par *model-checking* en suivant la méthodologie proposée dans (Coupat *et al.*, 2014). De plus, dans le cadre d'une utilisation du filtre logique en tant que contrôleur, la commande consistant à activer (respectivement désactiver) la sortie des vérins puis à les désactiver (respectivement activer) au cycle API suivant, est sûre de fonctionnement et permet de faire tourner sans risque 1, 2 ou 3 palettes dans le sens horaire.



$$CSs_1 = \uparrow Q1^+ \cdot \overline{b1} = \overline{Q1^{+*}} \cdot Q1^+ \cdot \overline{b1}$$
$$CSs_2 = \uparrow Q1^+ \cdot i1 \cdot i2 = \overline{Q1^{+*}} \cdot Q1^+ \cdot i1 \cdot i2$$
$$CSs_3 = \uparrow Q1^+ \cdot i1 \cdot \overline{b2} = \overline{Q1^{+*}}.\, Q1^+ \cdot i1 \cdot \overline{b2}$$
$$CSs_4 = \uparrow Q1^+ \cdot X31 = \overline{Q1^{+*}} \cdot Q1^+ \cdot X31$$
$$CSs_5 = \uparrow Q1^+ \cdot Q2^{+*} \cdot i1 = \overline{Q1^{+*}}.\, Q1^+ \cdot Q2^{+*} \cdot i1$$
$$CSs_6 = \downarrow Q1^+ \cdot \overline{h1} = Q1^{+*} \cdot \overline{Q1^+} \cdot \overline{h1}$$

$$CSc_1 = i1 \cdot \mathbf{Q1}^+ \cdot Q2^+$$
$$CSc_2 = i2 \cdot \mathbf{Q2}^+ \cdot Q3^+$$
$$CSc_3 = i3 \cdot \mathbf{Q3}^+ \cdot Q4^+$$
$$CSc_4 = i4 \cdot \mathbf{Q4}^+ \cdot Q1^+$$

Figure 9 : Sous-ensemble de contraintes de sécurité relatif au vérin 1

7 Conclusion

Cet article a présenté le principe, les applications et la mise en œuvre de la commande par contraintes développée par le CReSTIC de l'Université de Reims Champagne-Ardenne. Depuis les premiers travaux, l'objectif a été la proposition d'une méthode simple à mettre en œuvre, permettant de garantir la sécurité de la commande des API. Les 3 utilisations possibles : filtre bloquant, superviseur et contrôleur offrent des applications et des perspectives intéressantes pour la sûreté de fonctionnement des systèmes manufacturiers dont nous poursuivons actuellement le développement.

Références

Behrmann G., Bengtsson J., David A., Larsen K.G., Pettersson P., Yi W. (2002). Uppaal implementation secrets. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems.

Benlorhfar R., Annebicque D., Gellot F., Riera B. (2011). Robust filtering of PLC program for automated systems of production, 18th IFAC World Congress, Milano, Italy, august 2011.

Cassandras C. G., Lafortune S., (1999). Introduction to discrete event systems. Boston, MA: Kluwer Academic Publishers.

Coupat R., Meslay M., Burette M-A, Riera B., Philippot A., Annebicque D. (2014) Standardization and Safety Control Generation for SNCF Systems Engineer, 19th IFAC World Congress 2014 (IFAC WC), Cape Town, South Africa.

Fabian M., Hellgren A. (1998), "PLC-based implementation of supervisory control for discrete event systems," Decision and Control, 1998. Proceedings of the 37th IEEE Conference on , vol.3, no., pp.3305,3310 vol.3, 1998.

Hietter Y., Roussel J.-M., Lesage J.-J. (2008). Algebraic synthesis of dependable logic controllers 17th IFAC World Congress, Seoul (Korea), pp. 4132-4137, July.

IEC INTERNATIONAL STANDARD 61131-3 (2013). Programming languages for programmable logic controllers, CEI/IEC 61131-3: 2013.

Marangé P., Gellot F., Riera B., (2007). Remote control of automation systems for D.E.S. course, IEEE Transaction on Industrial Electronics, vol.54, no.6, pp.3103-3111. .

Marangé P., Debernard S., Gellot F., Pacaux M., Philippot A., Poulain T., Riera B. et Pétin J.-F. (2014), Approche de détection et d'explication d'erreur de commande par filtrage robuste. Hermès/Lavoisier, Journal Européen des Systèmes Automatisés, 48(4):339-372, décembre 2014. 10.3166/jesa.48.339-372

Ramadge G., Wonham W. M. (1989). The control of discrete event systems, Proc. IEEE, Special issue on DEDSs, 77, pp.81-98.

Riera B., Marangé P., Nocent O., Magalhaes A., Vigario B. (2009). Complementary usage of real and virtual manufacturing systems for safe PLC training, 8th IFAC Symposium on Advances in Control Education (ACE09), Kumamoto, Japon, October 2009.

Riera B., Coupat R., Philippot A., Gellot F., Annebicque D (2014). Control design pattern based on safety logical constraints for Manufacturing Systems: Application to a Palletizer, IFAC WOrshop on DES (WODES'14), Paris, France, May 2014.

Tournier L. (2005) Approximation of dynamical systems using s-systems theory: application to biological systems. Proc. of International Symposium on Symbolic and Algebraic Computation, Beijing, China, pages 317-324, 2005.