Filtre logique et Théorie du Contrôle Supervisé

Dimitri Renard, David Annebicque, Ramla Saddem, Bernard Riera

▶ **To cite this version:**

Dimitri Renard, David Annebicque, Ramla Saddem, Bernard Riera. Filtre logique et Théorie du Contrôle Supervisé. Modélisation des Systèmes Réactifs (MSR'25), CReSTIC, LICIIS, Nov 2025, Reims, France. ⟨hal-05491587⟩

HAL Id: hal-05491587

https://hal.science/hal-05491587v1

Submitted on 3 Feb 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY-NC-ND 4.0 - Attribution - Non-commercial use - No Derivative Works - International License

Filtre logique et Théorie du Contrôle Supervisé

Dimitri Renard[1,2*] David Annebicque[1], Ramla Saddem[1] and Bernard Riera[1]
[1] Université de Reims Champagne Ardenne, Reims, France
[2] Prosyst, Valenciennes, France
dimitri.renard@univ-reims.fr, david.annebicque@univ-reims.fr,
ramla.saddem@univ-reims.fr, bernard.riera@univ-reims.fr

Résumé

Cet article explore le lien entre un superviseur de coordination de tâches opératives synthétisé selon la Théorie du Contrôle Supervisé (SCT) et son pendant sous forme de filtres logiques. La SCT, bien que rigoureuse, est difficilement intégrable dans les automates programmables industriels (API) en raison de contraintes de synchronisation et de complexité. Les filtres logiques, plus simples à implémenter, offrent une alternative intuitive mais moins formalisée. Ce travail propose une démarche permettant d'exprimer les filtres logiques comme des formes réduites ou spécialisées de superviseurs SCT, conciliant ainsi rigueur théorique et applicabilité industrielle. Une application illustrative appuie la validité de l'approche proposée.

1 Introduction

L'Industrie 4.0 marque une nouvelle étape dans l'évolution des systèmes de production, caractérisée par l'intégration poussée des technologies numériques telles que l'Internet des objets (IoT), l'intelligence artificielle (IA), et le jumeau numérique. Cette transformation donne naissance à des usines intelligentes et interconnectées, où les équipements sont capables de communiquer entre eux, de prendre des décisions de manière autonome et de s'adapter en temps réel aux variations de l'environnement de production.

Dans ce contexte, les systèmes deviennent de plus en plus complexes. Les entreprises doivent répondre à des exigences croissantes en matière de flexibilité, de personnalisation de produits et de résilience face aux perturbations. Cela impose des méthodologies de conception du contrôle-commande à la fois robustes, sûres et intelligentes, afin d'assurer une production efficace et, surtout, sécurisée [1]. Cette exigence de sécurité devient particulièrement critique à mesure que des agents d'intelligence artificielle sont intégrés dans la boucle de décision des systèmes industriels.

La Théorie du Contrôle Supervisé (Supervisory Control Theory, SCT) [2] propose un cadre rigoureux fondé sur des modèles formels pour la synthèse de superviseurs capables d'imposer des spécifications comportementales sur des systèmes à événements discrets. Toutefois, malgré sa

puissance théorique, la SCT reste difficile à implémenter dans l'environnement industriel courant, notamment en raison de sa complexité algorithmique et de sa faible intégration avec les automates programmables industriels (API). Pour rappel, un API se caractérise par un fonctionnement cyclique et séquentiel avec 3 étapes. La première est la lecture des variables d'entrée (capteurs) et l'écriture dans la mémoire image. La deuxième est le traitement du programme écrit le plus souvent avec l'un des langages de la norme IEC 61131 [3]. Enfin, la troisième étape est la mise à jour des sorties. On notera que tous les calculs se font avec une valeur constante des variables d'entrée au moyen de la mémoire image.

Les filtres logiques, quant à eux, constituent une alternative plus simple et mieux adaptée aux contraintes industrielles. Bien que leur expressivité soit plus restreinte que celle des superviseurs issus de la SCT, ils offrent une implémentation directe, intuitive et compatible avec les architectures logiques standards utilisées dans l'industrie.

Ce travail propose d'explorer une démarche de représentation d'un superviseur, issue de la SCT, pour la coordination de tâches opératives sous forme de filtres logiques, dans le but de rapprocher ces deux approches. En démontrant leur compatibilité et leur fonctionnement équivalent, nous visons à faciliter la transition entre les nombreux travaux académiques de la SCT et la praticité d'implémentation des filtres logiques.

Cet article est structuré comme suit : après cette introduction, un état de l'art sera présenté, suivi de la méthodologie utilisée. Une application à visée pédagogique démontrera ensuite la méthodologie proposée. L'article se terminera par une discussion suivie d'une conclusion.

2 Etat de l'art

L'étude des filtres logiques et de la SCT constitue une base indispensable pour comprendre le rapprochement proposé dans ce travail. Ces deux approches, bien que développées dans des contextes différents, visent à garantir le respect de spécifications de sécurité ou de performance dans des systèmes de contrôle-commande, notamment dans les environnements industriels. Cette section présente une revue des contributions majeures dans chacun de ces domaines, en mettant en lumière leurs fondements, leurs applications et leurs limites respectives.

2.1 Filtre logique

Le concept de filtre logique dans les systèmes de contrôle-commande programmables, à l'origine, repose sur l'intégration d'un code correctif à la fin des programmes exécutés par les automates programmables industriels (API), afin d'assurer que leurs sorties soient toujours conformes aux contraintes de sécurité spécifiées [4]. Ce filtre fonctionne comme un modèle garantissant des propriétés de sécurité souhaitées, complétant ainsi le programme fonctionnel de contrôle-commande initial sans exiger de modifications sur celui-ci [5].

L'intérêt majeur de cette approche réside dans la séparation claire qu'elle instaure entre les aspects fonctionnels et sécuritaires. En effet, cette séparation facilite considérablement la conception et la maintenance des systèmes de contrôle-commande, puisque les contraintes de sécurité peuvent être définies et ajustées indépendamment des fonctions opérationnelles [6]. Le filtre logique intervient alors en temps réel pour corriger les sorties de l'API, évitant ainsi tous les comportements dangereux contraints par les spécifications. Cette solution est particulièrement pertinente lorsqu'il s'agit d'améliorer la sécurité des systèmes existants, même en présence de spécifications fonctionnelles incomplètes ou imprécises. Ainsi, le filtre logique représente une approche intuitive et efficace pour renforcer la fiabilité et la sécurité des systèmes industriels. Toutefois, il convient de vérifier rigoureusement la cohérence du filtre de manière formelle avant sa mise en œuvre opérationnelle [7].

La génération des filtres logiques par les méthodes de synthèse algébrique [8] se fonde sur l'utilisation de l'algèbre booléenne pour formaliser les exigences de sécurité et en déduire les règles de contrôle nécessaires [9]. Cette méthode consiste à transformer les contraintes de sécurité en équations algébriques, dont la résolution aboutit directement à la formulation du filtre logique sous forme algébrique. La synthèse algébrique se place dans le cas de l'algèbre des fonctions booléennes avec :

- $\Gamma = \{f : \mathbb{B}^n \to \mathbb{B}\}$ est l'ensemble des fonctions booléennes de dimension n
- $+$, $.$ et $^{-}$ sont les opérations OR, AND et NOT
- F et T sont les fonction respectivement toujours fausse et toujours vraie
- $(\Gamma, +, ., ^{-}, F, T)$ est l'algèbre de Boole des fonction booléennes
- $=$ représente l'égalité et $\leq$ l'inclusion entre deux fonctions booléennes

L'avantage de cette approche est de permettre un traitement rigoureux et systématique de spécifications complexes, garantissant ainsi une conformité stricte aux exigences de sécurité. La synthèse algébrique donne un cadre formel pour la génération de filtres logiques. Cependant, ce cadre ne couvre pas la partie relative à son utilisation.

2.2 Théorie du Contrôle Supervisé

La Théorie du Contrôle Supervisé (SCT), introduite par Ramadge et Wonham [2], constitue une approche rigoureuse fondée sur les systèmes à événements discrets (SED). Elle propose un cadre théorique pour la synthèse automatique de superviseurs capables de restreindre le comportement d'un système afin de satisfaire un ensemble de spécifications, tout en tenant compte de la présence d'événements non contrôlables [10].

Dans cette théorie, le système ("plant") est modélisé sous forme d'automates à états finis (FSM) pour exprimer l'ensemble des comportements réalisables. Un FSM G est défini par un 5-uplet $G = (Q, \Sigma, \delta, q_0, Q_m)$:

- Q : Ensemble des états de S.
- Σ: Ensemble des évènements de S (alphabet).
- δ : Fonction de transition. Il représente les transitions entre les états par les évènements

  $\delta : Q \times \Sigma \to Q$
- q_0: Etat initial de l'automate $q_0 \in Q$.
- Q_m: Ensemble des états marqués $Q_m \subseteq Q$.

La fonction de transition δ est étendue en une application (potentiellement partielle) $Q \times \Sigma^* \to Q$ en posant :

- $\delta(q, \epsilon) = q, q \in Q$
- $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)\ avec\ \sigma \in \Sigma, s \in \Sigma^*\ si\ q_1 = \delta(q_0, s)\ et\ q_0 = \delta(q_1, \sigma)\ sont\ définies$

Les contraintes devant être imposées au système sont également exprimées comme des langages formels ("spécifications"). La synthèse du superviseur consiste alors à déterminer un sous-langage contrôlable (et potentiellement non bloquant) respectant les spécifications. L'utilisation d'automates étendus (EFSM) intégrant des gardes et des actions, permet de faciliter la modélisation de

comportements plus riches. L'algorithme présenté dans [11] permet la transformation des EFSM en un ensemble de FSMs équivalents. Les algorithmes de la SCT sont donc compatibles avec cette approche de modélisation étendue.

La synthèse d'un superviseur peut être effectuée hors ligne, de manière optimale, en maximisant l'ensemble des comportements admissibles [12]. Pour les systèmes complexes, des approches en ligne, comme celles utilisant des fenêtres glissantes à N étapes, permettent une gestion dynamique et réactive des décisions [13].

De nombreux outils ont été développés pour faciliter la mise en œuvre de la SCT. Par exemple, Supremica [14] permet la modélisation, la vérification et la synthèse automatique de superviseurs, tandis que DEScMaker [15] réalise la transformation d'un projet Supremica en code Python, facilitant leur simulation ou intégration dans des prototypes.

Cependant, l'implémentation pratique de la SCT sur des API industriels reste complexe et un verrou à lever. Plusieurs obstacles se posent, notamment [16] :

- l'asynchronisme entre les événements du modèle et le synchronisme des signaux lors de la lecture des entrées de l'API,
- la gestion des choix multiples et de la causalité des événements,
- les problèmes de synchronisation exacte.

3 Méthodologie

La synchronisation de tâches opératives est un problème récurrent dans la commande logique par API que nous avons choisi de considérer dans cet article. Dans cette section, la démarche suivie pour la création d'un superviseur de coordination de tâches opératives puis sa transcription sous forme d'un filtre logique se comportant de manière identique seront décrites. Pour cela, la méthodologie proposera une modélisation du comportement du programme automate et des contraintes souhaitées sous forme d'automates à états dont résultera un superviseur.

3.1 Modélisation du superviseur de coordination

Pour réaliser la commande, nous proposons d'utiliser la méthodologie de conception proposée dans [17]. Elle décompose le développement de tâches opératives conçues de manière séparée puis une phase de coordination. La méthodologie proposée prend en entrée l'ensemble des tâches opératives développé pour le système, donne une modélisation en automate à états des tâches et des contraintes entre tâches pour générer un superviseur. Elle propose ensuite une implémentation dans un API en Python répondant à la norme IEC 61131 [3] en ajoutant une notion de priorité de tâche pour transformer le superviseur en contrôleur. La même démarche est réalisée pour créer une architecture de filtres logiques équivalente.

La Figure 1 montre une tâche opérative [18] exprimée en GRAFCET [19]. Par définition dans cette approche, une tâche opérative ne peut être stoppée pendant son exécution. En somme, une tâche opérative définie par construction son point de sortie impératif via sa condition finale. Cette manière de représenter la séquence par deux étapes dont une macro-étape permet une abstraction de la tâche opérative. Elle permet d'avoir deux transitions avec la condition initiale qui est commandable avec l'intégration lors de l'implémentation d'une variable d'autorisation. La condition finale est considérée comme incontrôlable car liée aux capteurs et observateurs du système. La transition de condition initiale et celle de condition finale ne peuvent pas être simultanées.

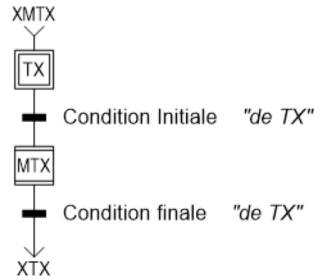

Figure 1 : Séquence générique

La phase de coordination est réalisée à la suite de la conception des tâches opératives. Cette phase répond à des contraintes provenant de la tâche elle-même. En effet, la tâche opérative a une condition initiale pour être autorisée. De plus, elle ne doit pas être déjà en cours pour pouvoir être déclenchée. A ces contraintes liées à la tâche, des contraintes entre tâches sont à ajouter. Celles-ci n'ont pas été prévues lors de leur conception séparée. C'est lors de la phase de coordination que ces contraintes doivent être ajoutées.

Pour réaliser ce contrôle de la coordination, la SCT est une méthode formelle pour contraindre les évènements commandables d'autorisation de tâches opératives. Pour cela, le programme API écrit sous forme de tâches opératives doit être modélisé sous forme de FSM ainsi que les contraintes entre tâches.

L'abstraction des tâches opératives est réalisée au moyen d'un automate à 2 états, l'un représentant le fait que la séquence est en attente, et l'autre, que la séquence est active comme représentée sur la Figure 2. Cette modélisation repose sur l'hypothèse que les tâches ne sont pas interruptibles ou forçables. La prise en compte des forçages ne fait pas l'objet de cet article.

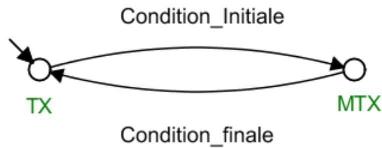

Figure 2 : Modélisation séquence

La condition initiale exprimée dans la spécification de la tâche opérative ne prend pas en compte la partie coordination. Il est donc nécessaire d'intégrer cette partie de coordination. Pour cela, la notion d'autorisation est ajoutée lors de l'implémentation dans la condition initiale. De plus, la condition finale prend en compte l'état final de la tâche opérative. C'est pour cela que la tâche X est autorisée par l'évènement AUT_T[X] (événement commandable) et se termine par l'évènement END_T[X] (événement non commandable).

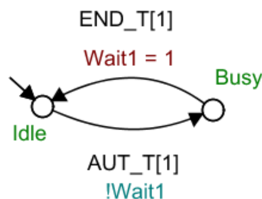

Figure 3 : Tâche avec informations de capteur

Lors de l'évènement de fin de tâche un ensemble d'informations de capteur est connu. En effet, la condition finale est validée lors de cet évènement. Des actions peuvent être ajoutées pour apporter de l'information dans le modèle "plant". Ces informations sont alors utilisables pour contraindre les conditions initiales intégrées dans les gardes des modèles de tâche. La Figure 3 représente une tâche opérative avec une garde pour son activation et une action lors de sa fin. Elle nécessite que le capteur "Wait1" représenté par une variable booléenne (transformée par la suite en FSM) ne soit pas activé pour être déclenchée (compris dans sa condition initiale). L'événement de fin de tâche nécessite que le capteur "Wait1" soit actif pour être déclenché. L'ensemble des informations disponibles en fin de chaque tâche permet de réduire la taille du superviseur synthétisé.

Avec la représentation proposée des tâches opératives sous forme d'EFSM, le non-parallélisme entre tâches se traduit par une interdiction de l'autorisation de tâche tant qu'une tâche incompatible est en cours.

Pour la représentation sous forme d'une spécification pour la synthèse de superviseur, la Figure 4 montre un exemple de contraintes entre deux tâches opératives. Ainsi, il est possible de représenter l'ensemble des contraintes entre tâches par l'ajout de ce canevas de spécification.

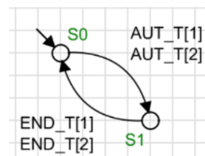

Figure 4 : Spécification de non-parallélisme entre tâche

Ce modèle peut être généralisé pour plus de contraintes avec l'ajout de n nouvelles tâches dans la liste des autorisations et des fins de tâche. Ainsi, l'autorisation d'une des n tâches interdit l'exécution en parallèle des $n-1$ autres. Le premier modèle de la Figure 5 représente un FSM interdisant l'exécution des tâches 1, 2 et 3 si l'une d'entre elles est en cours.

Le second FSM permet la généralisation pour l'exécution de t tâches en parallèle sur les n tâches avec $t < n$. Il se modélise sous la forme de t boucles. La Figure 5 montre un exemple avec 3 tâches. L'exécution en parallèle, deux à deux est autorisée, cependant l'exécution des 3 simultanément n'est pas possible.

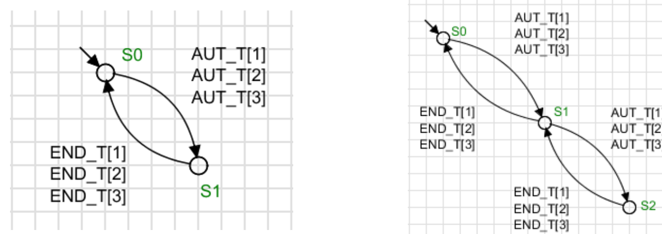

Figure 5 : Généralisation de la spécification de non-parallélisme entre tâches

Lors du démarrage du superviseur sur le système, les différents capteurs définis au travers de variables sont initialisés avec une valeur arbitraire. Pour initialiser les valeurs des variables de capteurs dans le superviseur le canevas de la Figure 6 est ajouté. Lors du premier cycle API, après la lecture des entrées, la valeur de chaque capteur est transformée en évènement pour faire évoluer le superviseur. L'ensemble des variables de capteur doit être initialisé avant qu'une autorisation de tâche puisse être déclenchée. Par la suite, le superviseur perçoit le système au travers des fins de tâches. Lorsqu'une intervention extérieure est possible dans le système, le modèle de tâche peut aussi

représenter la modification de la valeur d'un capteur. Sa priorité est alors maximale pour impacter les choix de tâche opérative lors du cycle en cours. Cependant, l'ajout de ces modèles supplémentaires augmente drastiquement la taille du superviseur. En effet, ces tâches n'ont pas de contrainte avec d'autres tâches. Ajouter un modèle, représentant un capteur tout ou rien, revient quasiment à doubler la taille du superviseur en termes d'états et de transitions.

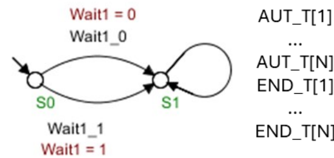

Figure 6 : Initialisation des capteurs dans le superviseur

3.2 Synthèse et implémentation de superviseur

A partir des modèles décrits dans la partie précédente, le superviseur est synthétisé à partir des algorithmes présents dans SUPREMICA [14]. Il en ressort un FSM. Celui-ci doit être transformé pour être implémenté dans un API. L'applicatif DEScMaker [15] est utilisé pour obtenir un superviseur en Python à partir du projet SUPREMICA. Ainsi, le superviseur pourra être intégré dans un API en langage Python. Des interfaces avec le superviseur permettent entre autres de le faire évoluer ou tester la validité d'un évènement dans un état.

La Figure 7 représente l'architecture d'utilisation du superviseur dans l'API. La condition finale de chaque tâche opérative est calculée en début de cycle après la lecture des entrées. Elle permet de faire évoluer les modèles de tâche “en cours“ se terminant dans leur état “en attente“. Les évènements de condition finale sont effectués les uns après les autres sur le superviseur pour le faire évoluer. L'ordre des événements de fin de tâche n'a pas d'importance sur le superviseur.

Dans l'ordre de priorité de déclenchement des tâches, l'ensemble des évènements d'autorisation sont testés sur le superviseur de manière séquentielle. Si l'évènement est autorisé dans l'état alors le superviseur évolue et la condition initiale sera franchie. Sinon l'évènement n'est pas pris en compte et la séquence ne sera pas autorisée lors de ce cycle. Avec ce principe, plusieurs autorisations de tâche peuvent être déclenchées lors d'un même cycle API. Après cette phase d'autorisation, les programmes des tâches en cours sont exécutés ainsi que le déclenchement des tâches autorisées lors de ce cycle API.

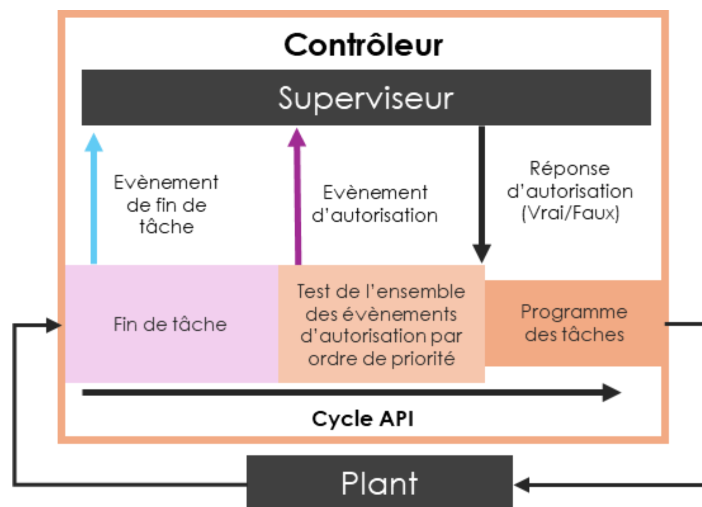

Figure 7: Architecture de déploiement du superviseur

Le projet d'implémentation (Figure 8) s'articule avec un programme Python (main) fonctionnant comme un API respectant la norme 61131[3]. Une librairie de communication permet la lecture des entrées (capteurs) et l'écriture des sorties (actionneurs) présentes dans le simulateur de Parties Opératives (Factory IO dans cette étude). Le superviseur généré par DEScMaker à partir du projet SUPREMICA est intégré dans le projet. Il comprend le superviseur ainsi que les fonctions permettant de le faire évoluer. Les interfaces du superviseur sont intégrées dans l'API pour tester les évènements et faire évoluer le superviseur.

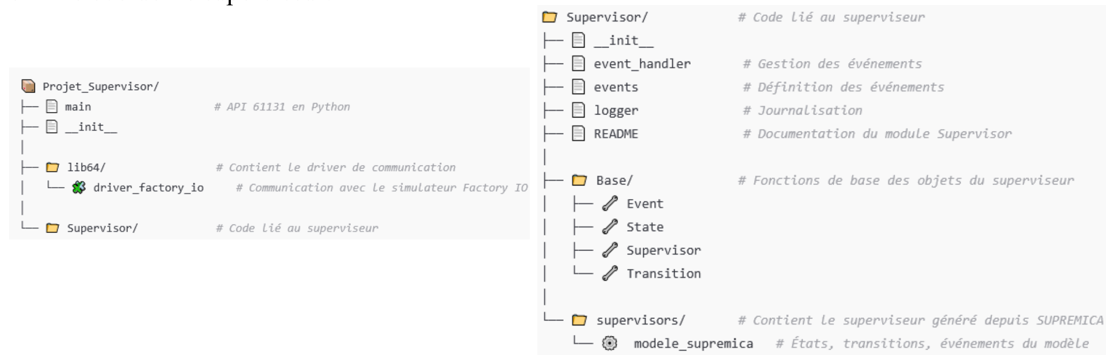

Figure 8 : Solution Python API et superviseur

3.3 Superviseur sous forme de filtre logique

L'implémentation de l'automate à états du superviseur dans l'API exige beaucoup de mémoire et est peu lisible pour l'utilisateur final. Une implémentation sous la forme d'un filtre logique garantissant les spécifications est davantage synthétique, léger à installer dans un API ce qui est clairement un atout. Toutefois, la transformation du superviseur en filtre logique n'est pas immédiate et nécessite de prendre en compte le fonctionnement séquentiel et cyclique de l'API. Pour y parvenir, sur la même base de spécifications, nous proposons de synthétiser un couple de filtres logiques.

Le filtre logique permet de garantir au contrôleur d'agir dans un espace d'états restreint par les différentes contraintes. Le modèle "plant" dans ce cas est le programme API. Les spécifications correspondent aux contraintes. Le programme API est composé de variables d'entrées, de sorties et internes. Le filtre logique se base sur une partie des variables internes et d'entrées pour valider ou non la commande souhaitée en fonction des contraintes exprimées. Tout type de variable peut être utilisé par l'intermédiaire de prédicats, élargissant le champ des possibilités au-delà des valeurs booléennes.

A partir de l'ensemble des contraintes intégrées pour le superviseur et transcrites en un ensemble de spécifications pour le filtre logique, nous pouvons synthétiser un filtre logique par synthèse algébrique. Le filtre logique a pour forme un ensemble d'équations logiques. Chaque équation correspond à un élément contrôlable du système (actionneur ou évènement interne). Il peut être directement implémenté dans un API.

La Figure 9 représente deux modèles de tâches avec leur produit synchrone puis avec l'ajout de la contrainte entre tâches d'un superviseur (identique au produit synchrone des 3 automates). Pour transformer ce superviseur en filtre logique, les autorisations de tâche (contrôlables) sont les variables impactées par le filtre. Les variables d'état sont utilisées dans les équations de ces variables. Dans les modèles, la variable X_i correspond à l'état initial de la tâche $X_i = \mathrm{IS_i}$. De plus, la tâche ne peut pas être dans l'état d'attente et en cours d'exécution. En conséquence, $X_i = \overline{Y_i}$. Ainsi, les états du modèle de la spécification de contrainte entre tâches peuvent s'exprimer sous la forme de :

$$F1 = X_1.X_2$$
$$F2 = \overline{X_1}.X_2 + \overline{X_2}.X_1$$

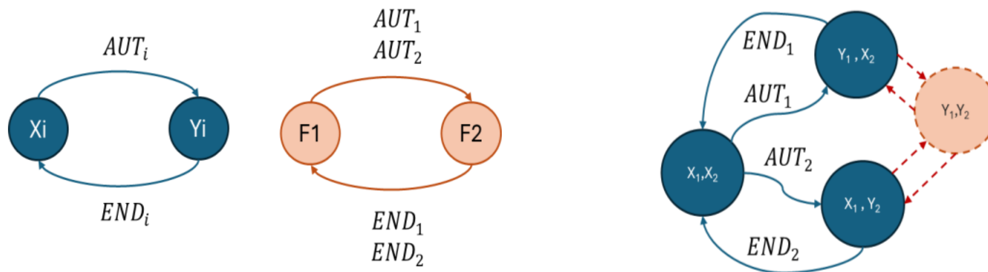

Figure 9 : Exemple de superviseur avec contrainte entre tâches

Les autorisations $AUTPotentielle_1$ et $AUTPotentielle_2$ impliquent $F1$et leur condition initiale respective (CI_X). En exprimant cette contrainte de cette manière, un temps de cycle API est nécessaire entre la fin de tâche et l'autorisation de la suivante. Pour pallier cela, les conditions finales sont calculées avant le filtre logique dans le cycle API. Elles sont donc déjà mises à jour en fonction du cycle API présent et utilisable dans le filtre pour connaître les tâches se terminant lors du cycle en cours. Ainsi :

$$AUTPotentielle_1 \leq X_1.CI_1.(X_2 + END_2)$$
$$AUTPotentielle_2 \leq X_2.CI_2.(X_1 + END_1)$$

Dans l'implémentation retenue pour la transformation d'une spécification en GRAFCET en code ST, une transition ne peut être évaluée comme vraie que si l'étape qui la précède est active. En particulier, la transition initiale ne peut être validée que si l'étape initiale est elle-même active. Par conséquent, cette transition ne peut être satisfaite lors du cycle final de la tâche opérative, puisque l'étape initiale n'est pas encore réactivée à ce moment-là du cycle. Ce comportement est géré par le superviseur, qui bloque toute émission d'un événement d'autorisation de tâche lorsque sa transition finale est active.

Cette implication n'est pas suffisante pour garantir le comportement. En effet, elle ne bloque pas le cas de la simultanéité dans ce filtre. Deux solutions sont possibles pour pallier ce cas, soit d'avoir un fonctionnement asynchrone avec une unique autorisation par cycle, soit en intégrant la contrainte de non-simultanéité dans le filtre logique. La synthèse algébrique offre un cadre formel pour spécifier et générer le filtre correspondant aux spécifications en prenant en compte les problématiques de simultanéité.

Pour avoir un comportement identique au superviseur, la solution proposée repose sur une architecture à deux filtres donnant des autorisations de façon asynchrone. Ainsi, les problèmes de simultanéité sont gérés par construction dans l'architecture et non directement dans les filtres logiques. L'architecture est représentée dans la Figure 10.

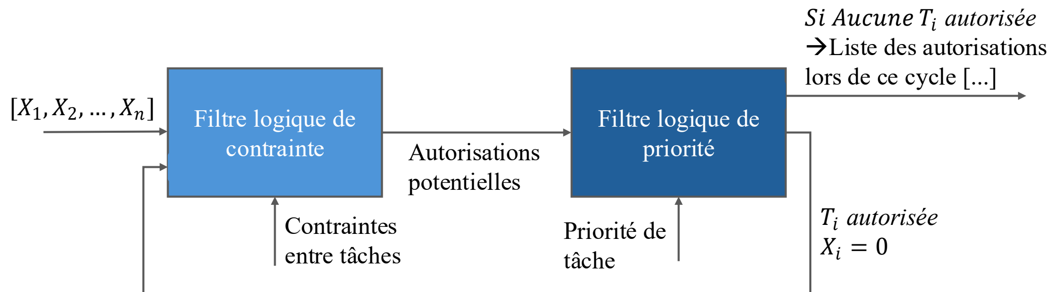

Figure 10: Architecture de l'implémentation des filtres logiques

Le filtre logique de contraintes a comme entrée l'ensemble des X_i de tâches. Il a été généré pour répondre à un ensemble de contraintes entre tâches. La simultanéité dans les contraintes de ce filtre n'est donc pas prise en compte. Les équations des $AUTPotentielle_X$, avec l'ensemble des contraintes avec les tâches C_1 à C_N sont de la forme :

$$AUTPotentielle_X = X_X . CI_X . \left(X_{C_1} + END_{C_1}\right) . (\dots) . \left(X_{C_N} + END_{C_N}\right)$$

La gestion de la simultanéité est réalisée dans le filtre logique de priorité. Ce filtre prend en entrée l'ensemble des autorisations potentielles pour en autoriser au plus qu'une. Ainsi, la simultanéité est contrainte directement par construction dans l'architecture proposée. Ce filtre est construit par une généralisation de la solution donnée par la synthèse algébrique aux spécifications suivantes. Pour 3 tâches opératives définies. Une autorisation AUT_X implique $AUTPotentielle_X$. De plus, l'ensemble des AUT_X s'excluent. Un ordre de priorité est connu (ici les AUT_X sont triés par ordre de priorité). Le système d'équations suivant correspond à cette description. La maximisation correspond à la priorisation.

$$\begin{cases} AUT_1 \leq AUTPotentielle_1 \\ AUT_2 \leq AUTPotentielle_2 \\ AUT_3 \leq AUTPotentielle_3 \\ AUT_1 . AUT_2 + AUT_1 . AUT_3 + AUT_2 . AUT_3 = 0 \end{cases}$$

$$Maximisation\ dans\ l'ordre\ AUT_1, AUT_2\ puis\ AUT_3$$

La solution à ce système d'équations est :

$$\begin{cases} AUT_1 = AUTPotentielle_1 \\ AUT_2 = AUTPotentielle_2 . \overline{(AUTPotentielle_1)} \\ AUT_3 = AUTPotentielle_3 . \overline{(AUTPotentielle_1 + AUTPotentielle_2)} \end{cases}$$

La solution pour construire le filtre de priorité peut être généralisée pour N tâches triées par ordre de priorité :

$$\begin{cases} AUT_1 = AUTPotentielle_1 \\ AUT_2 = AUTPotentielle_2 . \overline{(AUTPotentielle_1)} \\ \cdots \\ AUT_N = AUTPotentielle_N . \overline{(AUTPotentielle_1 + AUTPotentielle_2 + \cdots + AUTPotentielle_{N-1})} \end{cases}$$

Lors d'un cycle, si l'une des variables AUT_X est vraie, alors les autorisations des tâches moins prioritaires ne sont pas possibles par construction. La tâche la plus prioritaire ayant son autorisation potentielle est autorisée.

En bouclant ce principe avec la mise à 0 de la condition initiale (X_i) de la tâche qui a été autorisée, il est alors possible d'autoriser plusieurs tâches simultanément lors d'un cycle API sans prise en compte directement dans le filtre de contrainte. Lorsque aucune tâche supplémentaire ne peut être autorisée lors du cycle API en cours, l'ensemble des autorisations de tâche sélectionnées de façon asynchrone est alors déclenché. Cela se traduit par l'activation des fonctions de transition initiales des tâches opératives autorisées dans la suite du cycle API. Les tâches déclenchées dans les cycles précédents sont exécutées en même temps.

4 Application

Dans cette section, nous présentons une application visant à valider l'équivalence comportementale entre deux approches de supervision : d'une part, un filtre logique de sécurité généré par synthèse algébrique à partir de spécifications exprimées en logique booléenne, et d'autre part, un superviseur modélisé sous forme d'un FSM selon les principes de la Théorie du Contrôle Supervisé (SCT).

Le simulateur Factory IO est utilisé pour simuler un système physique [20]. Pour cette application, un système composé de 3 convoyeurs et d'un robot a été conçu (Figure 11). La commande est réalisée sous la forme de 13 tâches opératives, visant à acheminer les composants sur les deux convoyeurs d'entrée, puis à les évacuer sur le convoyeur de sortie via une transition effectuée par le robot. Entre ces tâches, 14 contraintes de non-parallélisme sont exprimées. La Table 1 répertorie les tâches opératives du système ainsi que les contraintes de parallélisme. L'ordre de priorité des tâches est : 1,3,5,6,9,10,12,7,11,2,4,8,13. La tâche T1 est la plus prioritaire, tandis que la tâche T13 est la moins prioritaire dans ce système. Ces priorités ont été définies de manière empirique, mais représentent le flux de pièces.

Table 1 : Tâches et contraintes

Tâche	Description	Contrainte de non-parallélisme entre tâches
T1	Acheminement pièce en position 1	T12
T2	Acheminement pièce en position 3	T13
T3	Rotation avec une pièce de la position 0 à 1	T6
T4	Rotation avec une pièce de la position 1 à 2	T12
T5	Déposer la pièce	T8, T9, T10, T11
T6	Rotation sans pièce de la position 0 à 3	T3
T7	Rotation avec une pièce de la position 3 à 2	T8, T13
T8	Rotation sans pièce de la position 2 à 0	T5, T7, T10, T11
T9	Evacuation d'une pièce	T5
T10	Rotation sans pièce de la position 2 à 1	T5, T8, T11
T11	Rotation sans pièce de la position 2 à 3	T5, T8, T10
T12	Prendre une pièce en position 1	T1, T4, T8
T13	Prendre une pièce en position 3	T2, T7, T8

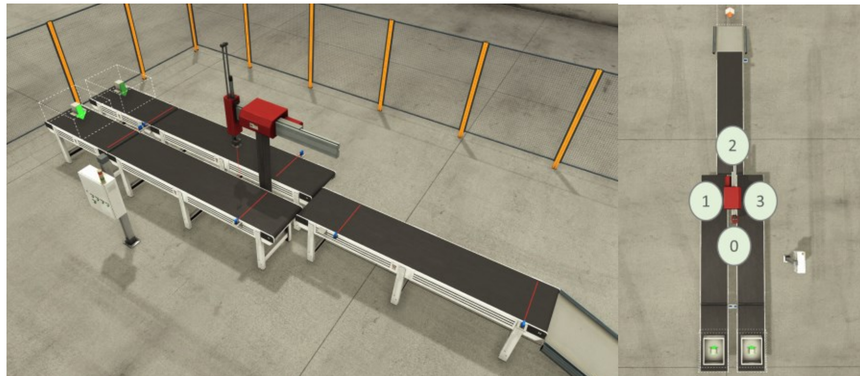

Figure 11 : Système d'application

Pour comparer l'approche par superviseur à celle par filtres logiques, l'architecture de la Figure 12 est mise en œuvre. A chaque cycle API, le résultat des autorisations provenant du superviseur est comparé à celui provenant des filtres logiques ayant les mêmes spécifications afin de vérifier qu'il existe une bisimulation. Si le résultat est différent entre les deux approches, un fichier de logs est généré.

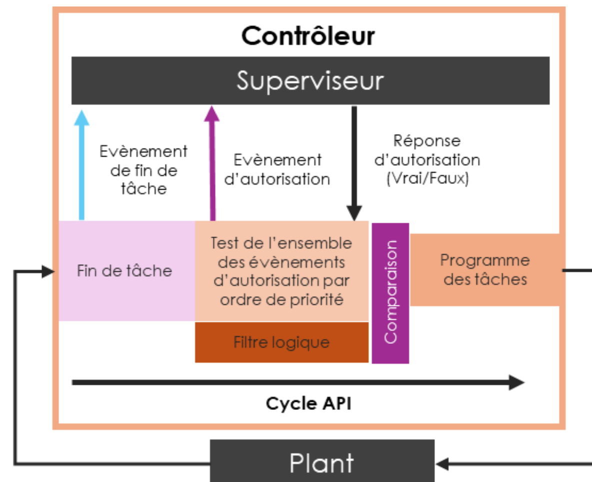

Figure 12 : Architecture de comparaison entre filtre logique et superviseur

Pour synthétiser le superviseur, les éléments présents dans la Figure 13 ont été déclarés. Les 13 tâches sont modélisées suivant le canevas de tâches. Les variables des 4 capteurs et de l'observateur de position du bras robotisé sont déclarées avec leur initialisation. Les contraintes entre tâches définies dans la Table 1 sont déclarées sous forme de spécifications. La synthèse du superviseur (équivalent dans ce cas au produit parallèle) donne un FSM de 664 états. Celui-ci est difficilement visualisable dû à sa taille. Le superviseur est transcrit en Python à l'aide de DEScMaker et implémenté dans un API en Python. En langage ST, le code est structuré en un *CASE* des 664 états. L'ordre de test des évènements sur le superviseur est réalisé suivant celui défini pour cette application. Si la transition finale d'une tâche est vraie alors son évènement d'autorisation n'est pas testé lors de ce cycle.

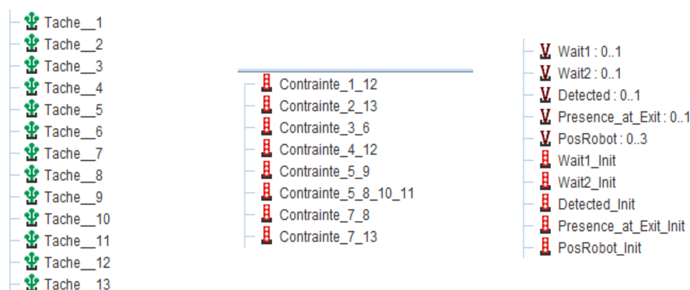

Figure 13: Plant et Spécifications

Les filtres logiques de contrainte et de priorité sont générés à partir des spécifications ainsi que de l'ordre de priorité. Le filtre logique de contrainte se traduit pour chaque tâche par une équation. Par exemple, les tâches T1 et T8 ont pour équations d'autorisation potentielle :

$$AutPotentielle_1 = X_1.(X_{12} + END_{12})$$

$$AutPotentielle_8 = X_8.(X_5 + END_5).(X_7 + END_7).(X_{10} + END_{10}).(X_{11} + END_{11})$$

Le filtre logique de priorité est construit à partir des priorités de tâche suivant la solution généralisée donnée par la synthèse algébrique. Un extrait du filtre est montré dans la Figure 14. Sa forme d'équation logique simplifie sa transposition entre les différents langages de programme. L'ordre de calcul des équations de ce filtre de priorité a une importance. Il est intégré dans le programme API tel que généré.

```
tache[1].Aut = tache[1].AutPotentielle
tache[3].Aut = tache[3].AutPotentielle and not tache[1].AutPotentielle
                    ...
tache[13].Aut = tache[13].AutPotentielle and not (tache[1].AutPotentielle or tache[3].AutPotentielle or ... or tache[8].AutPotentielle)
```

Figure 14: Filtre de priorité

Le superviseur et les filtres logiques sont tous implémentés dans l'API. La liste des autorisations de tâche validée par le superviseur est comparée à la liste des tâches validée par les filtres logiques à chaque cycle API. Les autorisations sont répercutées sur les programmes des tâches opératives exécutés à la suite de la phase d'autorisation réalisée en parallèle entre les deux approches. Elles permettent la commande du système commandé par le contrôle des actionneurs. En évoluant, les conditions du système changent, permettant la validation des conditions initiales de tâches pouvant alors être autorisées.

Par l'expérience menée sur ce système, aucun fichier de logs n'a été généré. A chaque cycle API, les autorisations de tâche issues des filtres logiques ont été identiques à celles retournées par le superviseur montrant qu'ils sont bisimilaires. Dans le cadre de cette application, le comportement des filtres logiques a donc été identique à celui du superviseur pour la coordination de tâches.

5 Discussion et conclusion

Les résultats obtenus confirment que la modélisation d'un superviseur SCT et la mise en œuvre de filtres logiques conduisent à un même comportement de contrôle-commande. Cette équivalence expérimentale illustre que le filtre logique agit comme une formulation condensée du superviseur, tout en offrant un niveau de garantie équivalent en termes de sécurité.

Au-delà de cette démonstration de faisabilité, l'approche présentée révèle plusieurs atouts majeurs pour son adoption en contexte industriel. D'une part, la traduction directe des contraintes de contrôle en langage ST (IEC 61131-3) [3] simplifie l'intégration dans les API tout en réduisant la complexité inhérente à la gestion d'un superviseur sous forme d'un FSM.

En somme, cette étude établit un pont concret entre la rigueur de la théorie du contrôle supervisé et les exigences opérationnelles des systèmes automatisés. La généralisation de cette méthodologie à des cas plus variés ainsi que l'exploration de vérification de propriétés (ex. : non-blocage) sur les filtres logiques sont des perspectives de recherche pertinentes pour tirer le meilleur parti de ces deux approches. L'utilisation de plusieurs filtres logiques de priorité pour plusieurs modes de fonctionnement du système est une autre perspective de recherche pour étendre cette approche pour des systèmes flexibles.

Références

[1] J. Zaytoon et B. Riera, « Synthesis and implementation of logic controllers – A review », *Annual Reviews in Control*, vol. 43, p. 152-168, 2017, doi: 10.1016/j.arcontrol.2017.03.004.

[2] P. J. G. Ramadge et W. M. Wonham, « The control of discrete event systems », *Proc. IEEE*, vol. 77, nº 1, p. 81-98, janv. 1989, doi: 10.1109/5.21072.

[3] International Electrotechnical Commission, *IEC 61131-3 Programmable controllers Part 3: Programming languages*, Geneva, Switzerland., 2013.
[4] B. Riera, P. Alexandre, D. Annebicque, et F. Gellot, « La commande par contraintes logiques de sécurité : principe, applications et mise en oeuvre », présenté à Modélisation des Systèmes Réactifs (MSR 2015), 2015. [En ligne]. Disponible sur: https://inria.hal.science/hal-01224261
[5] R. Pichard, « Contribution to the Control of discrete event systems by logical filter », PhD thesis, Université de Reims Champagne-Ardenne, Reims, 2018. [En ligne]. Disponible sur: https://theses.fr/2018REIMS025.pdf
[6] 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*, vol. 27, nº 4, p. 1659-1667, 2019, doi: 10.1109/TCST.2018.2827329.
[7] P. Marangé, « Synthèse et filtrage robuste de la commande pour des système manufacturiers sûrs de fonctionnement », PhD thesis, Université de Reims Champagne-Ardenne, Reims, 2008.
[8] Y. Hietter, « Synthèse algébrique de lois de commande pour les systèmes à évènements discrets logiques », PhD thesis, ENS Cachan, France, 2009.
[9] T. Ranger, A. Philippot, et B. Riera, « Algebraic Synthesis of Safety Logical Filter on Manufacturing Systems », *IFAC-PapersOnLine*, vol. 55, nº 2, p. 169-174, 2022, doi: 10.1016/j.ifacol.2022.04.188.
[10] C. G. Cassandras et S. Lafortune, *Introduction to discrete event systems*, Third edition. Cham: Springer, 2021.
[11] M. Sköldstam, K. Akesson, et M. Fabian, « Modeling of discrete event systems using finite automata with variables », présenté à 2007 46th IEEE Conference on Decision and Control, déc. 2007, p. 3387-3392. doi: 10.1109/CDC.2007.4434894.
[12] Y. Hu, D. Wang, M. Yang, et J. He, « Integrating reinforcement learning and supervisory control theory for optimal directed control of discrete-event systems », *Neurocomputing*, vol. 613, p. 128720, 2025, doi: 10.1016/j.neucom.2024.128720.
[13] R. C. Hill et S. Lafortune, « Scaling the formal synthesis of supervisory control software for multiple robot systems », présenté à 2017 American Control Conference (ACC), mai 2017, p. 3840-3847. doi: 10.23919/ACC.2017.7963543.
[14] R. Malik, K. Åkesson, H. Flordal, et M. Fabian, « Supremica–An Efficient Tool for Large-Scale Discrete Event Systems », *IFAC-PapersOnLine*, vol. 50, nº 1, p. 5794-5799, 2017, doi: 10.1016/j.ifacol.2017.08.427.
[15] T. Possato, J. H. Valentini, L. F. P. Southier, M. A. C. Barbosa, et M. Teixeira, « DEScMaker: a Tool for Automated Code Generation for Discrete Event Systems Controllers », *Science of Computer Programming*, p. 103350, 2025, doi: 10.1016/j.scico.2025.103350.
[16] M. Fabian et A. Hellgren, « PLC-based implementation of supervisory control for discrete event systems », présenté à Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171), déc. 1998, p. 3305-3310 vol.3. doi: 10.1109/CDC.1998.758209.
[17] D. Renard, D. Annebicque, R. Saddem, et B. Riera, « Novel Supervisor-Based Architecture for Logic Controller Design », présenté à 11th IFAC Conference on Manufacturing Modelling, Management and Control, Trondheim, Norvège, 2025.
[18] P. Taillard, « L'analyse stucturée par tâches », *technologie*, vol. 170, p. 38-43, 2010.
[19] International Electrotechnical Commission, *GRAFCET specification language for sequential function charts, IEC 60848*, IEC 60848, 2013.
[20] B. Riera *et al.*, « HOME I/O et FACTORY I/O : 2 logiciels innovants de simulation de PO pour la formation à l'automatique », présenté à Colloque consacré à l'Enseignement des Technologies et des Sciences de l'Information et des Systèmes (CETSIS), Le Mans, France, 2017. [En ligne]. Disponible sur: https://hal.science/hal-01882591