Détermination de l'équivalence comportementale d'algorithmes de contrôle - commande

Vincent Gourcuff, Olivier de Smet, Jean-Marc Faure

Détermination de l'équivalence comportementale d'algorithmes de contrôle - commande *

Vincent GOURCUFF, Olivier DE SMET et Jean-Marc FAURE
LURPA – EA 1385 – ENS de Cachan,
61 av. du Prés. Wilson, F-94235 Cachan Cedex, France
{gourcuff, de_smet, faure}@lurpa.ens-cachan.fr

1er février 2006

Résumé

La norme IEC 61508 sur la sécurité fonctionnelle des systèmes critiques préconise l'utilisation de méthodes formelles pour les logiciels de niveau SIL (Safety Integrated Level) 3 et 4. Cependant aucune précision sur l'utilisation de ces méthodes formelles dans le cycle de développement n'est indiquée. Le but général des travaux présentés dans cet article, et réalisés dans le cadre d'un contrat de recherche financé par Alstom Power Plant Information and Control Systems, est d'introduire des méthodes formelles pour aider à la réalisation d'applications de contrôle commande certifiées. Plus précisément cet article propose une méthode s'appuyant sur un outil de vérification formelle et permettant de vérifier que les représentations du même algorithme de commande dans des langages différents conduisent au même comportement si on se réfère aux entrées/sorties du système de commande. Cette méthode repose sur une représentation par automates à états traduisant, à un niveau d'abstraction adéquat, le comportement des langages d'automatismes du point de vue des entrées/sorties et traductible dans la syntaxe d'un outil de model - checking. Enfin, une application de cette méthode formelle dans le cadre industriel est présentée, montrant les résultats d'expériences sur un cas réel de contrôle de centrale thermique en utilisant l'outil de model - checking NuSMV et un outil de traduction automatique, développé dans le cadre de ces travaux.

Keywords : Programmable Logic Controllers, Behavioral equivalence, Dependability, Model Checking.
Mots-clés : Automate Programmable Industriel, Équivalence comportementale, Sûreté, Model Checking.

*Ces travaux ont été réalisés dans le cadre d'un projet de recherche financé par Alstom Power Plant Information and Control Systems, Engineering tools department.

1 Introduction

La norme IEC 61508 [IEC00] propose plusieurs méthodologies permettant d'améliorer la sécurité fonctionnelle des systèmes critiques. Pour ce qui concerne les logiciels utilisés dans ces systèmes, cette norme préconise d'utiliser des méthodes formelles si l'on souhaite atteindre les plus haut niveaux de sécurité, qualifiés de SIL (Safety Integrated Level) 3 et 4.

Cependant, ce document normatif n'indique pas le mode d'utilisation d'une méthode formelle, ni quelles sont les techniques à privilégier. L'objectif de nos recherche est de faciliter l'application de ces préconisations en nous limitant aux approches formelles de vérification, en plus particulièrement de model - checking symbolique.

Cet article présente donc une méthode permettant d'intégrer une telle approche dans le cycle de développement d'une application de contrôle - commande.

Après avoir présenté dans la première section la problématique industrielle à résoudre, la deuxième partie présente l'approche retenue. Cette approche est détaillée dans les troisième et quatrième parties. La cinquième partie présente une application de cette méthode dans un cas industriel. La conclusion et les travaux futurs font l'objet de la dernière partie.

2 Problématique industrielle

La division Plant information and Control systems de la société Alstom Power, propose des solutions de contrôle - commande pour systèmes de production d'énergie. Ces systèmes critiques sont pilotés par des contrôleurs industriels à base d'entrées/sorties (figure 1). Ces contrôleurs permettent de calculer cycliquement la prochaine valeur de chaque sortie en fonction de l'état des entrées.

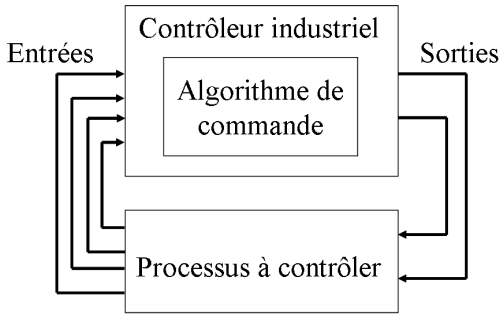

FIG. 1 – Principe d'un contrôleur industriel

Pour programmer ces contrôleurs industriels, l'entreprise Alstom a développé CONTROCAD, un outil intégré de développement d'applications de contrôle commande. A partir d'une programmation effectuée dans un des langages normalisés,

FBD ou SFC, issus de la norme IEC 61131-3 [IEC93], CONTROCAD permet de générer un programme sous forme de code compilé pour le contrôleur industriel (voir figure 2).

L'objectif final de nos travaux est de vérifier le code compilé par rapport aux spécifications de l'utilisateur décrit en FBD ou SFC. Pour ce faire, nous allons comparer le comportement de ces deux programmes dans tous les cas d'évolutions. Autrement dit, nous allons déterminer l'équivalence comportementale des deux programmes.

Dans cet article nous nous limitons à la méthode de comparaison des programmes écrits en LEA, un langage intermédiaire proche du langage ST de la norme IEC 61131-3, et en C. La raison est l'implantation déjà bien encrée de ces langages dans le développement des systèmes automatisés d'ALSTOM Power, autant en terme de culture des ingénieurs d'études que d'utilisations opérationnelles déja présentes dans de nombreux projets d'automatisation.

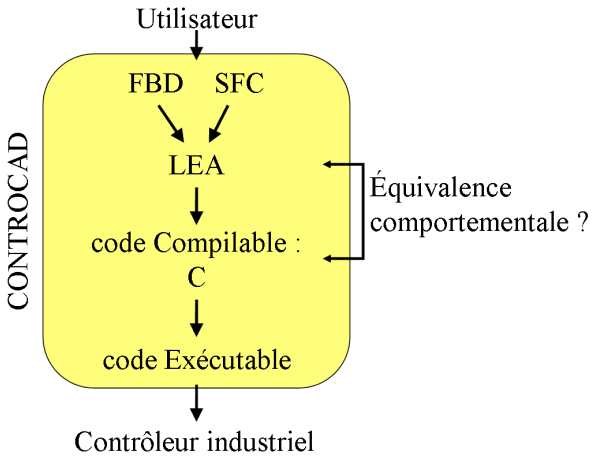

FIG. 2 – Génération de programme pour contrôleur industriel

3 Approche proposée

[Riv04] propose de certifier la compilation et les programmes compilés dans le cadre des programmes de contrôle aéronautiques. Ces travaux se placent dans le même cadre que nos recherches par rapport aux contraintes des normes aéronautiques. Son approche utilise une représentation des programmes basée sur les fonctions de transfert symbolique pour vérifier que la source et le programme compilé ont le même comportement. Ces fonctions de transfert sont soit concrètes (Translation Validation) soit abstraites (Invariant Translation) et permettent de vérifier des invariants précis au niveau du langage assembleur. Cependant cette méthode s'adresse aux programmes implantés sur des contrôleurs non-cycliques.

Dans notre cas, les contrôleurs industriels ont un comportement cyclique, représenté sur la figure 3. Cette exécution cyclique comporte trois étapes correspondant à :

- **La lecture des entrées :** une recopie de l'état des cartes d'entrées dans la mémoire du contrôleur industriel.
- **L'exécution du programme de contrôle :** une suite séquentielle d'affectation de variables.
- **L'émission des sorties :** une recopie de l'état de la mémoire du contrôleur industriel dans les cartes de sortie.

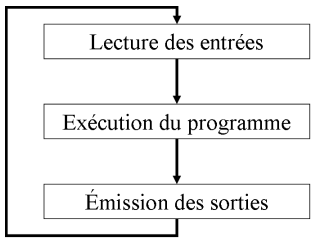

FIG. 3 – Cycle d'exécution d'un programme sur un contrôleur industriel

Vis à vis de cette exécution, la seule étape pouvant générer un état dangereux est l'émission des sorties, c'est à dire l'étape où le contrôleur industriel influence le processus critique. Donc le seul comportement à vérifier est celui des sorties du contrôleur industriel en fonction de ses entrées. En effet, d'un point de vue extérieur, un contrôleur industriel ne fait que le lien entre les entrées et les sorties ; la méthode d'obtention de cette relation peut être abstraite car elle n'influe pas sur le processus à contrôler. Donc la vérification se résume à déterminer si, pour un même ensemble de variables d'entrée et pour toutes les séquences possibles de ces variables d'entrée, les algorithmes émettent les mêmes sorties, si leur état initial est identique.

Notre expérience dans le cadre de la vérification nous a permis de déterminer que l'utilisation du model - checking, décrite dans [SBB+99], peut être une solution pour déterminer cette équivalence comportementale. Notre méthode, présenté figure 4, consiste d'abord à modéliser les différents programmes en prenant en compte le comportement cyclique du contrôleur. Une fois les modèles obtenus, l'utilisation d'un model - checker permettra de vérifier les critères d'équivalence comportementale. Ces critères se traduisent par la propriété suivante pour chaque variable de sortie : *Le long de tous les chemins d'exécution, tous les états vérifient l'égalité des valeurs de la variable de sortie calculée par chacun des programmes.* Si $sortie1\_LEA$ et $sortie1\_C$ sont les deux valeurs d'une variable $sortie1$ calculées respectivement par les programmes LEA et C, la traduction en logique temporelle CTL de la propriété pour cette variable est :
$AG(sortie1\_LEA = sortie1\_C)$

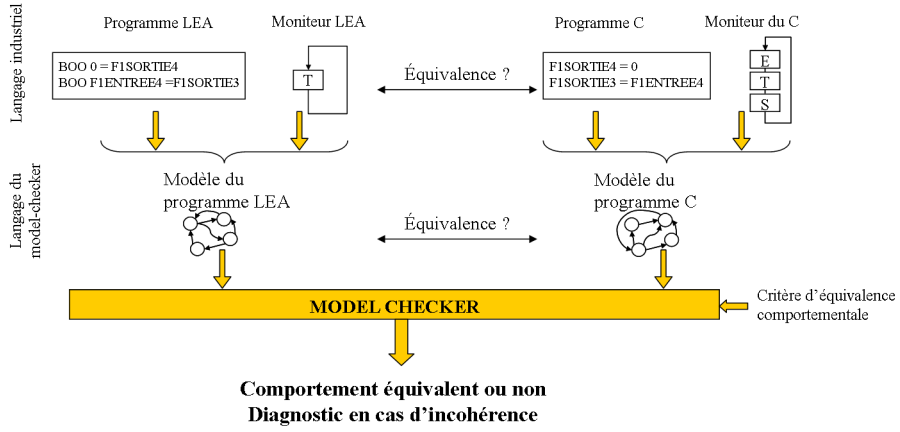

FIG. 4 – Approche de model-checking pour la détermination de l'équivalence comportementale

4 Modélisation d'algorithme de commande

Dans cette partie, nous développerons les règles de traduction des langages industriels vers le langage de l'outil de model-checking. Dans ce cadre, nous étudierons d'abord comment prendre en compte le comportement cyclique d'un contrôleur industriel dans les modèles.

4.1 Approche

Comme présenté dans [LCRRL99], plusieurs travaux ont permis de prendre en compte cette séquence d'exécution dans les outils de model - checking. En effet, le principal problème est de traduire le comportement des contrôleurs ainsi que le programme de commande dans un langage compatible avec les outils de model - checking.

Notamment, [RdSLC$^+$00] modélise les programmes de commande par un produit cartésien de l'état des variables du programme et de l'état d'évolution du programme en lui-même. L'idée principale de son travail a été de prendre en compte l'état du contrôleur industriel en plus de l'état des variables. Ainsi, en spécifiant à l'outil de model-checking à quel endroit du cycle il se trouve, il est contraint de n'effectuer qu'une opération par évolution. Chaque primitive du langage est exprimée par un automate à état, et une composition d'automates permet de modéliser un réseau entier.

Cette méthode a été conçue pour la modélisation de programmes multi-langages. Elle utilise la modularité d'un langage constitué d'un nombre fini de primitives. Elle a de plus l'avantage de modéliser finement le comportement réel du contrôleur industriel. Toutefois, le model-checker doit exécuter un pas de calcul à chaque

instruction traitée. De ce fait, le nombre d'états atteignables par le modèle du programme de commande implanté dans l'outil de model-checking augmente fortement avec le nombre de contacts, bobines ou boites fonctionnelles utilisées. Ce qui rend cette méthode inapplicable pour des programmes réels.

Mais dans notre cas, seul l'état des variables de sortie en fin de cycle est important. A partir de cette idée, [Moo94] propose une technique de modélisation des programmes Ladder Diagram. Étant parmi les précurseurs de ce type d'approche, son modèle ne tient compte que des quelques primitives du Ladder Diagram les plus utilisées : le contact, le contact inversé et la bobine. Ces primitives pouvant être directement transcrites en éléments d'équation booléenne, il lui est possible de faire une transposition immédiate du réseau en équation booléenne.

De plus, tenant compte du fait que la valeur d'une variable précédemment calculée dans le cycle a changé au moment de son évaluation lors d'un réseau postérieur, il remplace cette variable par la valeur qui lui a été affectée lors de son évaluation. Ainsi, il obtient un code pouvant s'exécuter en parallèle et s'affranchit des problèmes de séquentialité. Toutefois, sa méthode reste trop restrictive pour être applicable, mais elle est une base de l'approche par réécriture sous forme d'équations récurrentes.

Une extension de cette méthode est présentée dans [Car04] qui définit comment modéliser le comportement séquentiel d'un programme écrit en Ladder Diagram en un système d'équations récurrentes. Nous développerons ce concept pour l'appliquer à nos algorithmes de programmation industriels.

4.2 Notation et sémantique.

Nos modèles utiliseront une sémantique spécifique qui permet de représenter l'affectation de chaque variable à l'aide d'un triplet (X, I, T) où :

- X est l'ensemble de variables x (représentant les entrées, sorties et variables internes) ;
- $I(X)$ est la valeur initiale de chaque variable x ;
- $T(X)$ définit le prochain état de chaque variable x.

Les opérations possibles pour définir $T(X)$ sont :

- les opérations booléennes des langages de la norme IEC 61131-3 sur les booléens : *et, ou* et *non* représentés respectivement par &, | et *!* ;
- une structure conditionnelle sous forme de tests en cascade.

Les opérandes utilisés par $T(X)$ peuvent être :

- l'appel des valeurs courantes des variables sorties ;
- l'appel des valeurs prochaines des variables sorties ou des variables d'entrée ;
- les valeurs 0 et 1.

$T(X)$ est donc un système d'équations logiques récurrentes d'ordre un qui représente la fonction de passage d'un cycle du contrôleur industriel à un autre.

4.3 Règles de modélisation.

Frontière de modélisation. Chaque programme de commande est considéré de manière indépendante et toutes les variables utilisées dans ce programme sont considérées comme variables d'état du modèle. En effet, nous considérons l'algorithme de commande et son implantation sur le contrôleur comme l'objet à modéliser. Les entrées sont considérées, au même titre que les sorties, comme des variables affectées par une équation récurrente.
Le modèle est donc composé de deux parties :

- **La lecture des entrées.** Elle se modélise par l'affectation des entrées à une valeur non-déterministe, 0 ou 1 dans notre cas booléen. C'est à dire, étant donnée une entrée *e* :
  $e \in X, T(e) := \{0, 1\}$.
- **Le calcul des sorties.** Chaque sortie ou variable interne est associée à une variable et sa valeur future est définie par une équation récurrente, c'est à dire une fonction dépendante des autres variables et d'elle-même. Plus formellement, étant donnée sortie *s* :
  $s \in X, T(s) := f(X, T(X))$.

Les affectations. Afin de modéliser le comportement séquentiel de l'algorithme de commande, il est nécessaire de différencier les valeurs avant affectation et après affectation. En effet, une variable peut être utilisée avant et après son affectation. Par exemple, si une variable *a* dépend de la variable *b*, il faut déterminer quand *a* est affectée dans l'algorithme par rapport à *b*.
Si la variable *b* n'est pas affectée avant la variable *a*, alors il faut utiliser la valeur courante de *b* :
$T(a) := b$
$T(b) := 1$
Si la variable *b* est affectée avant la variable *a*, alors il faut utiliser la valeur future de *b*, $T(b)$:
$T(a) := T(b)$
$T(b) := 1$

La règle précédente est respectée de la même manière avec les entrées. En effet, comme les entrées sont affectées à chaque début de cycle, leur utilisation s'effectue toujours avec leur valeur future. Étant donnée e une entrée et a une sortie :
$T(e) := \{0, 1\}$
$T(a) := T(e)$

Les affectations multiples. Un algorithme de commande peut permettre des affectations multiples d'une même variable. Dans ce cas, seule la dernière affectation est mémorisée pour l'émission finale des sorties. Cependant, cet état transitoire de la variable peut être utilisé comme paramètre d'une autre variable ou d'elle-même. Pour représenter cet état transitoire dans le modèle une variable supplémentaire est introduite. Dans l'exemple suivant (figure 5), la variable a est affectée plusieurs

fois. Mais seule la dernière affectation est prise en compte pour l'émission de la sortie. Cependant son état intermédiaire est utilisé pour le calcul de la variable b. Il faut donc le mémoriser, ici dans la variable a_0.

programme du contrôleur	modèle en équations récurrentes
$c = a;$	$T(c) := a;$
$a = 0;$	$T(a_0) := 0;$
$b = !a;$	$T(b) := !T(a_0);$
$a = b;$	$T(a) := T(b);$

FIG. 5 – Modélisation d'une affectation multiple

Structures conditionnelles. Les langages industriels ont des structures conditionnelles simples, de type 'if-then-else'. Celles-ci sont modélisées par des tests en cascade pour chacune des variables affectées dans ces structures conditionnelles.

programme du contrôleur	modèle en équations récurrentes
	$T(e) := \{0,1\}$
$a = e$	$T(a) := e$
$If(a)$	$T(b) := case$
$b = 1$	$T(a) : 1;$
	$!T(a) : b;$
	$esac;$

FIG. 6 – Modélisation des structures conditionnelle

Nous pouvons remarquer que l'affectation qui est conditionnelle en C devient systématique dans notre modélisation. A chaque cycle d'évolution de notre modèle, une variable est toujours calculée du fait de l'utilisation des équations récurrentes. Si ce n'est pas le cas dans l'algorithme modélisé, alors, dans notre modélisation, la future valeur de la variable est égale à sa valeur actuelle. Si une structure conditionnelle permet l'affectation de plusieurs variables, il faut découpler ces affectations afin d'obtenir une structure conditionnelle pour chaque variable.

Abstraction du temps. Certaines parties de l'algorithme de commande comprennent de blocs de commande dépendant du temps physique. Dans ce cas, le temps physique est abstrait de façon logique. Par exemple, pour une temporisation cela se résume à savoir si le délai est écoulé ou non. Nous pouvons illustrer cette abstraction sur un exemple avec une temporisation à l'enclenchement, ou Time ON

delay (TON). Son comportement est décrit dans la norme IEC 611313 [IEC93] et permet de retarder l'enclenchement d'une sortie Q par rapport à son entrée In (voir figure 7) . Nous pouvons remarquer que la temporisation à enclenchement

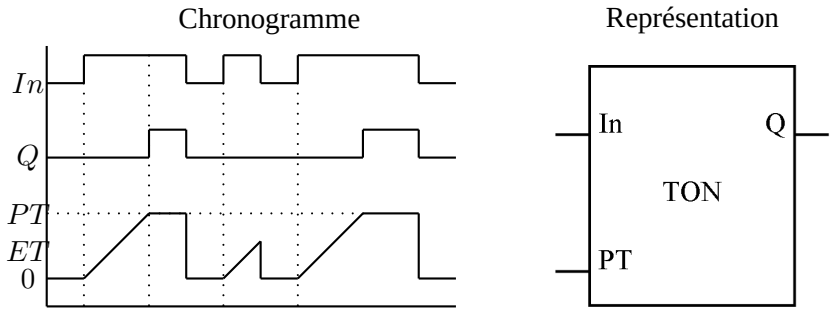

FIG. 7 – Temporisation à l'enclenchement

comporte trois états logiques, d'un point de vue des états des sorties et des entrées :

- la temporisation est inactive : In=0 donc Q=0
- la temporisation est enclenchée : In=1 et Q=0
- la temporisation est déclenchée : In=1 et Q=1

La modélisation logique du temps reprend la modélisation réalisée par [RS00], elle ne tient compte que de ces trois états, comme présentés dans la figure 8. Nous

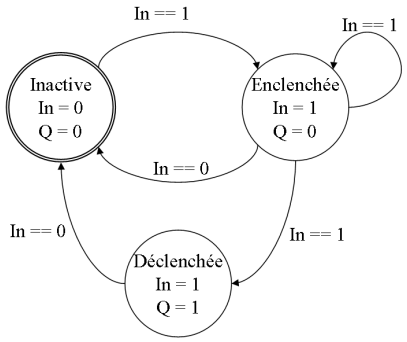

FIG. 8 – Modèle à trois états d'une temporisation

pouvons voir que la principale différence avec les affectations des variables booléennes est l'introduction de l'indéterminisme au niveau de la transition *enclenchée* vers *déclenchée*. Cet indéterminisme peut être traduit sous la forme d'une équation générique caractérisant toutes les évolutions possibles du modèle, comme présenté

ci-dessous :

$$
\begin{aligned}
T(Q) := \quad & case \\
& \quad In : 0; \\
& \quad In\&!Q : \{0,1\}; \\
& \quad In\&Q : 1; \\
& esac;
\end{aligned}
$$

Cette abstraction peut être étendue à tous les blocs fonctionnels utilisant du temps physique. Cette généralisation des abstractions à tous les blocs fonctionnels utilisés dans les programmes industriels fera l'objet de travaux futurs.

Exemple. La figure 9 présente un exemple d'utilisation de ces règles de modélisation sur un fragment de programme écrit en LEA avec deux entrées, $INIT$ et $ENTREE5$, et deux sorties, ${}_{5}6RS_{S}TA$ et $SORTIE4$.

5 Optimisation des modèles obtenus

Cette section présente une méthode d'utilisation et d'optimisation des modèles présentés dans la section précédente.

Principe général. Une fois les modèles de chaque algorithme de contrôle - commande obtenus, une synchronisation doit être réalisée afin de vérifier l'équivalence comportementale. Pour se faire, les entrées sont communes aux deux modèles alors que les sorties sont différenciées mais doivent avoir la même valeur initiale. Nous pouvons remarquer que ces valeurs initiales ne sont pas fixées, le nombre d'états initiaux dépend donc du nombre de sorties. Donc le temps de vérification final dépend, mise à part le nombre d'états atteignables du programme, du nombre de sorties. Pour remédier à cette possible explosion combinatoire, le paragraphe suivant propose une méthode de réduction du nombre de variables de sortie à initialiser.

Restriction aux variables séquentielles. Jusqu'à présent toutes les variables de sortie sont affectées par des équations récurrentes. Or nous pouvons remarquer que certaines variables n'ont pas besoin de ce concept de récurrence pour être calculées. Nous allons donc distinguer deux types de variables :

Variable séquentielle : variable dépendant d'elle-même ou utilisée avant d'être affectée.

Variable non-séquentielle : variable dont la valeur au cycle précédent n'est pas utilisée.

L'histoire des variables non-séquentielles n'a pas besoin d'être mémorisée car elle n'est jamais utilisée. De ce constat, une abstraction peut être réalisée afin de restreindre la représentation de l'algorithme aux variables séquentielles. Cette restriction nécessite plusieurs opérations :

Programme du contrôleur en LEA

```
IF INIT
THEN
    BOO  0  =  SORTIE4 =  _56RS_STA
ELSE
    IF  ENTREE5
    THEN
        BOO 0 =  SORTIE4
    ELSE
    BOO  _56RS_STA =  SORTIE4
    ENDIF
    BOO  SORTIE4 =  _56RS_STA
ENDIF
BOO / SORTIE4 =  SORTIE4
```

Modèle en équations récurrentes

```
T( SORTIE4_0) :=
        case
            INI : 0;
            !INI :
                case
                    T( ENTREE5) : 0;
                    !T( ENTREE5) :  _56RS_STA;
                esac;
        esac;

T( SORTIE4) :=
        !(case
            INI : 0;
            !INI :
                case
                    T( ENTREE5) : 0;
                    !T( ENTREE5) : _56RS_STA;
                esac;
        esac);

T( _56RS_STA) :=
        case
            INI : 0;
            !INI :
                case
                    INI : 0;
                    !INI :
                        case
                            T( ENTREE5) : 0;
                            !T( ENTREE5) : _56RS_STA;
                        esac;
                esac;
     esac;
```

FIG. 9 – Exemple de modélisation d'un code LEA

- les variables séquentielles sont identifiées ;
- les lectures de variables non-séquentielles dans les équations récurrentes sont remplacées par leur équation ; ceci est réalisé récursivement tant que les variables séquentielles dépendent de variables non-séquentielles ;

- les affectations des variables non-séquentielles sont retirées du modèle.

Cette restriction n'influence pas la vérification car l'équivalence comportementale des variables non-séquentielles peut toujours être déterminée en utilisant leur définition booléenne dans les propriétés correspondantes.

Exemple. L'exemple de la figure 10 présente une application de la méthode, présentée sur un petit modèle.

ancien modèle	modèle optimisé
$T(E) := \{0,1\}$	$T(E) := \{0,1\}$
$T(A) := T(E)$	$A := 0$
$T(B) := case$	$T(B) := case$
$T(A) : 1;$	$T(E) : 1;$
$!T(A) : B;$	$!T(E) : B;$
$esac;$	$esac;$

FIG. 10 – Exemple d'optimisation de modèle

6 Application

Cette section présente une application de la détermination de l'équivalence comportementale sur une application industrielle.

6.1 Présentation

Pour répondre aux contraintes de la norme IEC 61508, la méthode de détermination de l'équivalence comportementale a été appliquée aux algorithmes de commande écrits en LEA et en C, comme indiqué à la section 2.

L'exemple étudié concerne le contrôle d'une centrale thermique et comprend :

- 175 programmes de contrôle - commande, vérifiés indépendamment.
- 1857 variables de sorties affectées, dont 59 variables entières ignorées dans le cadre de cet article.

La vérification par model-checking est réalisée par l'outil NuSMV [CCG+02]. NuSMV est un model-checker symbolique acceptant la définition des modèles par ensemble d'équations récurrentes.

6.2 Automatisation de la traduction

Intérêt. Afin d'intégrer la vérification dans l'outil industriel CONTROCAD, un outil de modélisation automatique traduisant les différents algorithmes de commande en modèles compatibles avec le model checker NuSMV a été développé en

appliquant toutes les règles de modélisation décrites dans les sections précédentes. Cet outil prend comme entrée les deux codes et génère en sortie une réponse traduisant l'équivalence comportementale.

Principe. Les différentes étapes de la modélisation automatique sont présentées dans la figure 11. Cet outil a été développé en langage Python [Lut96] en utilisant le module de déclaration de grammaire TPG parser, basé sur les grammaires DCG (Definite Clause Grammars) définies par [PW80] pour l'analyse syntaxique des programmes de contrôle - commande.

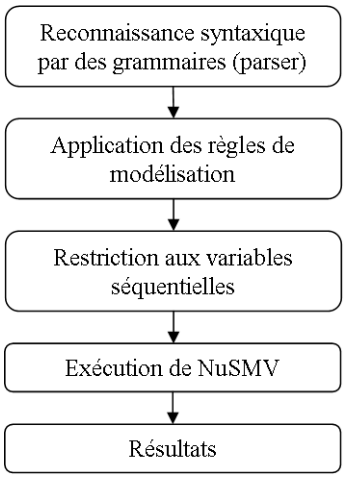

FIG. 11 – Étapes du traducteur automatique

Plusieurs points peuvent être remarqués quant à cette organisation :

- Les grammaires permettent reconnaître syntaxiquement les programmes, mais aussi limitent les possibilités du langage utilisé, comme préconisé dans la norme IEC 61508, et identifient les erreurs syntaxiques du code.
- Les temporisations sont représentées par des appels de fonction en LEA mais sont réalisées avec des compteurs associés au temps en C. Afin d'effectuer correctement les abstractions, le comportement du code C de la temporisation a été vérifié une seule fois et elles sont maintenant analysées par la grammaire comme un appel de fonction puis modélisé comme présenté dans la section 4.3.
- Lors de la génération du modèle NuSMV, les sorties LEA et C sont différenciées alors que les entrées sont déclarées communes.
- Les résultats de la vérification par NuSMV sont analysés et synthétisés afin de déterminer l'équivalence comportementale globale des deux programmes.

6.3 Résultats

Le tableau 1 présente les résultats obtenus dans le cas d'étude indiqué.

	modèle sans optimisation	modèle avec optimisation
Nombre de variables de sortie	1857	1857
Nombre de variables mémorisées	1857	636
Somme des états atteignables	7.922820061e+028	7.922820059e+028
Temps de traduction automatique	1 min	1 min
Temps de vérification sur Pentium 2.6 GHz	49.7 s	11.52 s

TAB. 1 – Performance de la vérification de l'application industrielle

Nous pouvons remarquer la diminution du nombre de variables mémorisées par le model-checker. Ceci permet un gain de temps de déclaration, de création et d'initialisation des variables et donc, globalement, un gain de temps de vérification.

Cependant le nombre de d'états atteignables varie peu entre le modèle non optimisé et le modèle optimisé. Cette variation est due aux variations du nombre d'états initiaux qui dépend directement du nombre de variables déclarées et mémorisées. La conservation de tous les états atteignables de chaque modèle est préférable dans notre abstraction afin de pouvoir déterminer dans tous les cas d'évolution l'équivalence comportementale des deux programmes.

7 Conclusion

Cet article présente une méthode de détermination de l'équivalence comportementale de deux programmes de contrôle - commande dans un cadre industriel. Cette méthode utilise le model-checker NuSMV pour effectuer une vérification des modèles de programmes de contrôle - commande. La principale difficulté d'une telle vérification est le choix des règles de transformation de l'aspect séquentiel de l'exécution du programme en automate à état fini compatible avec le model-checker. Le choix retenu est la modélisation en équations récurrentes qui permet d'obtenir la seule information nécessaire : le comportement des sorties du programme de contrôle - commande.

Grâce à une série de règles de transformation et à une optimisation par réduction du nombre de variables à mémoriser, la vérification est efficace, réalisée dans des temps raisonnables et correspond aux attentes de l'entreprise vis à vis des contraintes de la norme IEC 61508 et des contraintes de performances.

Cependant, cette détermination de l'équivalence comportementale est limitée, pour l'instant, aux variables booléennes. Les travaux futurs permettront la prise en compte des variables entières et réelles utilisées dans les programmes. Puis cette

vérification sera généralisée aux autres langages afin de valider l'ensemble du cycle de développement des algorithmes de commande.

Références

[Car04] O. Cardin. Apport de la réécriture pour la vérification d'un programme de commande par model-checking. Rapport de DEA Production Automatisée, ENS Cachan, 2004.

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

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

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

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

[Lut96] M. Lutz. *Programming Python*. O'Reilly, 1996.

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

[PW80] F. Pereira and D. Warren. Definite clause grammars for language analysis–a survey of the formalism and a comparison with augmented transition networks. In *Artificial Intelligence*, 1980.

[RdSLC+00] O. Rossi, O. de Smet, S. Lampérière-Couffin, J.-J. Lesage, H. Papini, and H. Guennec. Formal verification : a tool to improve the safety of control systems. In *4th Symposium on Fault Detection, Supervision and Safety for Technical Processes (IFAC Safeprocess 2000), Budapest, Hungary*, pages 885–890, 2000.

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

[RS00] O. Rossi and Ph. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proc. 4th Int. Conf. Automation of Mixed Processes : Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany, Sept. 2000*, pages 177–182. Shaker Verlag, Aachen, Germany, 2000.

[SBB+99] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, A. Petit, et al. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 1999.