Un cadre pour la vérification automatique de programmes IL

G. Canet[1,3], B. Denis[2], A. Petit[3], O. Rossi[2] et Ph. Schnoebelen[3]
(1) Alcatel CIT, Corporate Research Center
Route de Nozay – 91461 Marcoussis Cedex
(2) LURPA – ENS de Cachan
61 av. Pdt. Wilson – 94235 Cachan Cedex
(3) LSV – UMR 8643 CNRS & ENS de Cachan
61 av. Pdt. Wilson – 94235 Cachan Cedex

Résumé:
Nous nous intéressons à la vérification automatique de programmes d'API (automates programmables industriels) écrits dans le langage IL (Instruction List), un des cinq langages définis par la norme IEC 61131-3. Nous proposons une sémantique formelle d'un sous-ensemble représentatif du langage IL et un codage direct de cette sémantique dans un outil de model-checking. Nous vérifions ensuite automatiquement des propriétés comportementales riches exprimées dans une logique temporelle linéaire. La pertinence de notre approche est illustrée par le traitement de l'exemple d'un chariot automoteur sur rail.

Mots-Clés:
Génie informatique, vérification, Instruction List, méthodes formelles, automates programmables

I Introduction

Les API (automates programmables industriels) sont souvent utilisés dans des domaines critiques et l'élaboration de programmes sûrs et fiables revêt une importance grandissante. La taille et la complexité de ces programmes rendent indispensable l'utilisation d'outils automatisés pour aider à garantir les propriétés de sécurité.

Nous nous intéressons ici à l'un des cinq langages définis par la norme IEC 61131-3 [1] : le langage textuel IL (pour Instruction List). Dans cette étude préliminaire de faisabilité, nous nous limitons aux programmes IL que nous appellons *simples*, c'est-à-dire n'autorisant que des variables de type « entier borné » ou « booléen » et aucune temporisation.

Nous proposons une sémantique formelle d'un programme IL simple sous forme d'un système de transitions. Sommairement, un état représente les valeurs des différentes variables à un point donné de l'exécution du programme. Une transition correspond à l'exécution d'une ligne de programme. Nous exprimons par ailleurs les propriétés comportementales attendues, formalisées à partir du cahier des charges, par des formules de la logique temporelle LTL (*Linear Temporal Logic*, [3]). Enfin un traducteur exprime le système de transitions dans le langage d'entrée du model-checker *Cadence SMV*[1] [11], qui effectue ensuite automatiquement la vérification des propriétés.

Nous illustrons notre approche en traitant l'exemple d'un chariot automoteur sur rail. Diverses propriétés attendues du système sont vérifiées ; nous montrons entre autres comment des propriétés comportementales peuvent être traitées.

L'originalité de notre approche, qui permet de valider des programmes d'API existants, réside essentiellement dans le fait qu'elle combine à la fois (1) une sémantique formelle complète, mais néanmoins relativement simple d'un sous-ensemble intéressant du langage IL, (2) un codage direct de cette sémantique dans un outil de model-checking, permettant (3) la vérification automatique de propriétés comportementales complexes. Si diverses modélisations de programmes IL à base d'automates temporisés [10], de réseaux de Petri [6] et [5]), de logiques d'ordre supérieur [15], de langages synchrones [7] ou de systèmes conditions/événements ([4] et [8]) ont déjà été proposées [9], notre approche présente l'avantage de permettre de traiter entièrement des cas réels de taille non élémentaire.

II Définition et sémantique opérationnelle du langage IL

II.1 Définition d'un programme IL

La syntaxe du langage IL est proche de l'assembleur. Un programme IL est composé d'une déclaration des variables et d'un corps de programme contenant les instructions (cf. l'exemple de la section III).

II.1.1 Déclaration des variables:

La déclaration d'une variable est la donnée d'un symbole x désignant la variable, de son type $\mathrm{t}(x)$ et d'une valeur initiale optionnelle $\mathrm{init}(x)$. Nous nous limiterons ici aux cas des types booléens et entiers bornés. Un type entier borné t est défini par deux entiers min_t et max_t avec $\texttt{minint} \leq min_t \leq max_t \leq \texttt{maxint}$ où `minint` et `maxint` sont des constantes.

[1] Nous avons choisi ce model-checker en raison de la portée de ses techniques symboliques, mais notre approche pourrait être adaptée à tout autre model-checker traitant les systèmes de transitions.

II.1.2 Corps du programme:

Le corps d'un programme IL est une suite finie de lignes de commande. Chacune de ces lignes de commande est un couple (l_i, ins_i), formé d'une étiquette facultative et d'une instruction. Une instruction est formée d'un opérateur et éventuellement d'un argument.

Les opérateurs IL permettent d'effectuer certaines opérations sur les variables du programme, la plupart du temps en passant par l'accumulateur. Nous citons quelques instructions significatives :

`LD x`	place la valeur de x dans l'accumulateur (`LDN` place $\neg x$) ;
`ST x`	place dans x la valeur de l'accumulateur ;
`S x`	place la valeur logique *Vrai* dans x si l'accumulateur contient une valeur vraie ;
`R x`	place la valeur logique *Faux* dans x si l'accumulateur contient une valeur vraie ;
`AND x`	calcule un *et* logique entre l'accumulateur et x et place le résultat dans l'accumulateur (`ANDN` calcule un *et* entre l'accumulateur et $\neg x$) ;
`EQ x`	teste l'égalité de l'accumulateur et de x et place le résultat (booléen) dans l'accumulateur ;
`LT x`	teste si l'accumulateur (entier borné) est strictement inférieur à x (entier borné) et place le résultat dans l'accumulateur ;
`JMP l`	effectue un saut vers l'étiquette l (`JMPC` : faire le saut seulement si l'accumulateur vaut *Vrai*, `JMPCN` seulement si l'accumulateur vaut *Faux*) ;
`RET`	termine l'exécution du programme (`RETC` termine si l'accumulateur vaut *Vrai*, `RETCN` si l'accumulateur vaut *Faux*) ;
`ADD`	effectue l'addition entière entre l'accumulateur et x et place le résultat dans l'accumulateur.

Il existe quelques règles simples (déclaration unique, respect du typage...) définissant quels programmes sont bien formés, et un compilateur IL exige que ces règles soient suivies. Nous ne les détaillons pas ici et supposons par la suite que nos programmes sont bien formés.

II.2 Sémantique opérationnelle

On considère un programme IL constitué de k lignes. On note Σ l'ensemble des noms de variables de ce programme et E le sous-ensemble de Σ constitué des noms de variables d'entrée. Une valuation de Σ est une fonction associant à chaque variable x sa valeur, c'est-à-dire *Faux* ou *Vrai* si x est de type booléen, et un élément de $[min_{\mathrm{t}(x)}, max_{\mathrm{t}(x)}]$ si x est une variable de type entier borné.

II.2.1 Comportement cyclique:

Les API que nous considérons dans cet article fonctionnent de manière cyclique. Un cycle est composé de trois phases :

1. acquisition des entrées ;
2. exécution du programme ;
3. affectation des sorties.

Notre modèle traduit la dynamique interne de l'API. Nous ne représentons pas explicitement la phase d'affectation des sorties, mais nous distinguons une phase de *fin de cycle*, qui indique que le programme est arrivé en fin d'exécution. C'est la phase où les variables de sortie peuvent être observées. Cette phase précède la phase d'acquisition des entrées. Après l'acquisition des entrées a lieu une nouvelle exécution du programme.

II.2.2 Système de transitions associé à un programme IL:

On représente le comportement du programme IL par un système de transitions $S = \langle Q, \rightarrow, I \rangle$, où l'ensemble des états (aussi appelés *configurations*) Q est constitué des triplets de la forme (V, a, pc) avec :

- V est une valuation de Σ ;
- $a \in \{Faux, Vrai\} \cup [\mathtt{minint}, \mathtt{maxint}]$ est la valeur de l'accumulateur ;
- $pc \in \{1, \ldots, k+1\} \cup \{err\}$ est le compteur de programme.

$I \subseteq Q$ est l'ensemble des *états initiaux* : (V, a, pc) est initial si et seulement si le compteur de programme pc vaut 1 et si $V(x) = \mathrm{init}(x)$ pour toute variable x dont la valeur initiale a été déclarée.

Un état de la forme (V, a, pc) avec $1 \leq pc \leq k$ correspond à l'état du système avant exécution de la ligne pc du programme IL. Les états de la forme $(V, a, k+1)$ sont ceux de fin de cycle.

Chaque ligne de programme donne lieu à un ensemble de transitions. Pour la ligne i, ces transitions s'appliquent aux états (V, a, i) où le pointeur de programme est sur la ligne i. En général, l'état courant est modifié en fonction de la nature de l'instruction et on passe à la ligne $i+1$.

Nous donnons ici quelques exemples significatifs des différents cas, en notant ins_i l'instruction de la ligne i de notre programme IL.

$ins_i = \mathtt{ST}\ s$ **avec** s **de type booléen:**

- pour tout $a \in \{Faux, Vrai\}$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = a$.
- pour tout $a \notin \{Faux, Vrai\}$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V, a, err)$.

$ins_i = \mathtt{ST}\ s$ **avec** s **de type** $\mathrm{t}(s)$ **entier borné:**

- pour tout $a \in [min_{\mathrm{t}(s)}, max_{\mathrm{t}(s)}]$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = a$;
- pour tout $a \notin [min_{\mathrm{t}(s)}, max_{\mathrm{t}(s)}]$ (ceci couvre à la fois les cas où l'accumulateur contient une valeur booléenne et ceux où il contient une variable entière hors de l'intervalle de définition de s), il existe une transition $(V, a, i) \rightarrow (V, a, err)$.

$ins_i = \texttt{AND}\ s$ **avec** s **de type booléen:**

- pour tout $a \in \{Faux, Vrai\}$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V, a \wedge V(s), i+1)$;
- pour tout $a \notin \{Faux, Vrai\}$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V, a, err)$.

$ins_i = \texttt{ADD}\ s$ **avec** s **de type** $\mathrm{t}(s)$ **entier borné:**

- pour tout $a \in [\texttt{minint}, \texttt{maxint}]$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V, a + V(s), i+1)$ si $a + V(s) \in [\texttt{minint}, \texttt{maxint}]$ et $(V, a, i) \rightarrow (V, a, err)$ sinon ;
- pour tout $a \in \{Faux, Vrai\}$ et toute valuation V, il existe une transition $(V, a, i) \rightarrow (V, a, err)$.

$ins_i = \texttt{JMP}\ l$ **avec** l **étiquette de la ligne** i_l**:**
Pour tout a et toute valuation V, on crée une transition $(V, a, i) \rightarrow (V, a, i_l)$.

Pour compléter ces transitions interprétant les instructions IL, il reste à définir le comportement du programme dans les états d'erreur et dans les phases d'acquisition des entrées.

Erreur:
L'état d'erreur est bloquant. Nous modélisons le blocage par une boucle infinie dans l'état d'erreur : pour tous V et a il existe une transition $(V, a, err) \rightarrow (V, a, err)$.

Lecture des entrées:
Enfin il existe des transitions modélisant la phase de lecture des entrées. Les nouvelles valeurs des entrées peuvent prendre des valeurs arbitraires (respectant le typage) : pour toutes valuations V et V' ne différant (éventuellement) que sur les variables d'entrée (i.e. si $x \in \Sigma \setminus E$, alors $V'(x) = V(x)$) il existe une transition $(V, a, k+1) \rightarrow (V', a, 1)$.

On observera que pour la relation $\rightarrow$ chaque état de Q est suivi d'un et d'un seul état, sauf dans le cas des états de fin de cycle où la lecture des entrées introduit du non-déterminisme.

III Exemple

III.1 Description

Nous prenons comme exemple un chariot automoteur de manutention, se déplaçant le long d'un rail porteur. Seul le sous-ensemble de translation sur le rail sera étudié ici. Ce sous-ensemble (cf. fig 1) se compose d'un chariot animé par un moto-réducteur, d'un pupitre comportant un clavier permettant de saisir un numéro de position, d'un voyant lumineux ainsi que trois boutons-poussoirs monostables.

La position du chariot sur le rail est élaborée par un codeur rotatif absolu qui fournit l'information sous la forme d'un entier codé sur 5 bits, allant de 0 à 31. Les variables entières seront donc toutes bornées par l'intervalle $[0, 31]$.

Le déplacement du chariot vers une position donnée est obtenu en saisissant un entier à l'aide du clavier et en validant la requête à l'aide du bouton go.

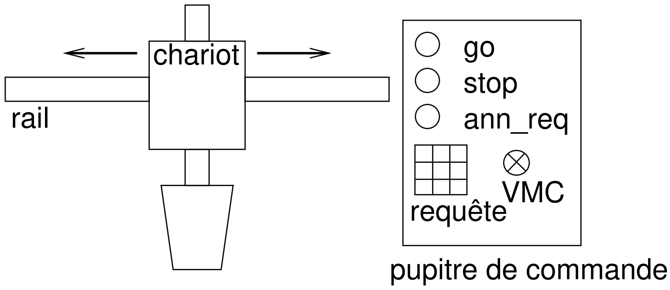

Fig. 1 – Chariot automoteur de manutention.

III.1.1 Entrées:

- pos (entier) est la position courante du chariot ;
- requête (entier) est la valeur tapée au clavier ;
- trois entrées booléennes go, stop, et ann_req, reliées à leurs boutons-poussoirs respectifs.

III.1.2 Sorties:

- MG : sortie booléenne qui correspond à un mouvement du moto-réducteur vers la gauche ;
- MD : sortie booléenne qui correspond à un mouvement du moto-réducteur vers la droite ;
- VMC : sortie booléenne qui correspond à l'allumage du voyant chariot.

III.1.3 Variables internes:

- mouvement (booléen) indique que le chariot est en mouvement ;
- dest (entier) est la mission en cours ;
- mission_en_cours (booléen).

III.2 Cahier des charges

Le comportement attendu du chariot est décrit par les règles suivantes :

- lorsque le moteur est à l'arrêt, il peut être mis en déplacement par un appui sur le bouton go, si la requête est différente de la position. Au début du déplacement le voyant s'allume ;
- le moteur est arrêté par un appui sur le bouton stop. Si une mission est en cours, le voyant reste allumé ;
- la mission s'arrête si le chariot est arrivé en position demandée par la requête ou le bouton ann_req est appuyé quand le chariot est à l'arrêt ;
- à la fin de la mission, le moteur s'arrête et le voyant s'éteint ;
- si on appuie simultanément sur plusieurs boutons, le bouton stop est prioritaire. Si stop n'est pas appuyé, aucun effet n'est produit ;
- pendant la marche du moteur, un appui sur le bouton go ne produit aucun effet ;
- pour des raisons d'inertie mécanique et de bon fonctionnement moteur, tout changement de direction du moteur doit passer par une phase d'arrêt.

III.3 Programme

Nous disposons d'un programme, écrit en IL, pour implémenter la commande du chariot. Le corps de ce programme, sans les déclarations de variables, est le suivant :

```
1           LD    stop
            JMPC  arret
```

```
                LD    dest
                EQ    pos
5               AND   mouvement
                JMPC  fin_mission
                LDN   mouvement
                ANDN  go
                AND   ann_req
10              JMPC  fin_mission
                LDN   mouvement
                AND   go
                ANDN  ann_req
                RETC
15              LDN   mission_en_cours
                JMPCN marche
                LD    1
                ST    mission_en_cours
                ST    VMC
20              LD    requete
                ST    dest
marche:         LD    pos
                EQ    dest
                RETC
25              LD    1
                ST    mouvement
                LD    pos
                LT    dest
                JMPC  droite
gauche:         LD    1
31              ST    MG
                RET
droite:         LD    1
                ST    MD
35              RET
fin_mission:    LD    0
                ST    mission_en_cours
                ST    VMC
arret:          LD    0
40              ST    MG
                ST    MD
                ST    mouvement
```

IV Vérification d'un programme par les méthodes du model-checking

IV.1 Codage de la sémantique opérationnelle

Nous utilisons la technologie du model-checking ([2], [12], [14]) pour vérifier les propriétés comportementales du système. Pour vérifier automatiquement un système par la méthode du model-checking, on en construit une modélisation formelle, puis on énonce formellement les propriétés à vérifier en utilisant un langage de spécification de propriétés, par exemple une logique temporelle. Un algorithme implanté dans un *model-checker* est capable de dire si le système vérifie ou non les propriétés énoncées. La plupart des model-checkers sont capables de fournir un diagnostic complétant la vérification dans le cas d'une propriété non vérifiée Par exemple, le model-checker propose un exemple d'exécution du système violant la propriété.

Lors de cette étude, c'est l'outil Cadence SMV [11] qui a été retenu mais d'autres model-checkers peuvent être utilisés. Le principal avantage de SMV est son utilisation de techniques puissantes de représentation symbolique, ce qui lui permet de vérifier facilement des systèmes de grande taille.

Le système de transition S associé à un programme IL est codé presque littéralement dans le langage du model-checker. Le plus délicat est de coder la relation de transition en décrivant l'évolution des variables. Nous avons réalisé un traducteur qui donne, pour tout programme IL simple, le codage pour SMV du système de transition S associé au programme.

Dans notre codage, en plus des variables déclarées par le programme, nous déclarons des variables représentant l'accumulateur et la ligne courante. La variable `pc`, correspondant à la prochaine ligne à exécuter, est une variable entière dont la valeur s'étend de 0 à 43 dans notre exemple où le programme compte 42 lignes. La valeur 0 correspond à l'état d'erreur.

Par exemple, l'évolution de la variable `mouvement` se décrit comme suit dans la syntaxe de SMV :

```
next (mouvement) := switch (pc)
  {
    26 : a ;
    42 : a ;
    default : mouvement ;
  } ;
```

Cela s'interprète de la manière suivante : pour chaque transition $(V, a, i) \to (V', a', i')$ de la sémantique opérationnelle, on a :

- $V'(\texttt{mouvement}) = a$ si $i = 26$ ou si $i = 42$;
- $V'(\texttt{mouvement}) = V(\texttt{mouvement})$ sinon.

Les valeurs de 26 et 42 correspondent au début des transitions correspondant aux lignes contenant `ST mouvement`.

Les variables d'entrée n'évoluent qu'au cours de la transition d'acquisition. Par exemple, l'évolution de la variable `requete` se définit de la manière suivante :

```
next (requete) := switch (pc)
  {
    0..42 : requete ;
  } ;
```

On notera que rien n'est spécifié dans le cas où `pc` vaut 43 : par défaut, SMV considérera que toutes les valeurs sont possibles.

IV.2 Vérification de propriétés

Nous décrirons les propriétés en LTL (cf. [2], [3], [14]), logique temporelle permettant d'exprimer des propriétés sur des événements, et sur l'enchaînement d'événements au cours d'une exécution du programme.

Dans cette section nous montrons comment énoncer formellement quelques propriétés typiques attendues du programme IL donné plus haut. Ces propriétés sont pour la plupart issues du cahier des charges.

IV.2.1 Cohérence de la commande du moteur:

Nous voulons vérifier que le moteur n'est jamais actionné dans les deux sens à la fois. Autrement dit, à aucun moment de l'exécution du programme, les variables MG et MD ne sont égales à *Vrai* toutes les deux. Cela s'énonce au moyen de la formule suivante :

$$\mathsf{G}\,\neg(\mathtt{MG} \wedge \mathtt{MD})$$

où le G signifie « dans tous les états de toutes les exécutions ».

Cette propriété est directement soumise au model-checker. SMV effectue un examen exhaustif des comportements du programme, et répond que la propriété est fausse. La trace fournie par SMV permet de détecter une erreur dans le programme : la partie du programme commandant l'activation du moteur gauche (lignes 30 à 32) est exécutée à tort. L'erreur se trouve à la ligne 14, qui devrait contenir l'instruction `RETCN` (au lieu de `RETC`).

Après correction du programme, la propriété est vérifiée par SMV.

IV.2.2 Respect de l'arrêt:

Nous vérifions que le moteur s'arrête si le bouton `stop` est actionné. Formellement, cette propriété peut s'énoncer par la formule suivante :

$$\mathsf{G}\,(\mathtt{stop} \rightarrow (\neg\mathtt{MG} \wedge \neg\mathtt{MD}))$$

qui se lit « *pour tout état de toute exécution, si la variable stop vaut vrai alors ni MG ni MD ne sont à vrai* ».

SMV nous répond que la propriété n'est pas vérifiée. L'explication tient au fait qu'il existe un délai entre le moment où le programme constate que le bouton `stop` est enfoncé (juste après une phase de lecture des entrées) et la mise à zéro des variables `MG` et `MD` (l'arrêt des moteurs, ligne 40 et 41).

Nous sommes dans une situation où la propriété attendue a été formalisée d'une façon trop stricte. En réalité, le seul moment où il est pertinent d'observer les variables de sortie représentant la commande des moteurs est la phase de fin de cycle. Si nous appelons **fdc** la proposition représentant la fin de cycle (dans notre cas **fdc** a la valeur de $(\mathtt{pc} = 43)$), nous pouvons énoncer la propriété qui nous intéresse par la formule

$$\mathsf{G}\,(\mathtt{stop} \rightarrow (\mathbf{fdc} \rightarrow (\neg\mathtt{MG} \wedge \neg\mathtt{MD})))$$

SMV nous répond que cette propriété est vérifiée par notre programme.

Il serait possible de vérifier la formule

$$\mathsf{G}\,(\mathtt{stop} \rightarrow \mathsf{F}\,(\neg\mathtt{MG} \wedge \neg\mathtt{MD}))$$

qui énonce que, si le bouton stop est actionné, les moteurs finiront inévitablement par s'arrêter (un jour). Cette propriété est moins précise que la précédente, et ne permet pas d'affirmer quoi que ce soit sur le délai nécessaire à la coupure du moteur.

IV.2.3 Non-alternance de la commande du moteur:

Nous vérifions que tout changement de direction du moteur passe obligatoirement par une phase d'arrêt. Formellement LTL exprime de telles propriétés au moyen de la formule

$$\mathsf{G}\,(\mathtt{MG} \rightarrow (\neg\mathtt{MD}\;\mathsf{W}\;(\neg\mathtt{MG} \wedge \neg\mathtt{MD})))$$

pour le passage de `MG` à `MD` (ainsi qu'une formule symétrique pour l'autre cas).

Cette formule se lit « *pour tout état de chaque exécution, si MG est activé, alors MD reste constamment désactivé jusqu'à ce que, éventuellement, le moteur soit à l'arrêt* ». Nous utilisons ici le combinateur W, ou *weak until*, qui s'exprime dans la syntaxe de SMV au moyen du combinateur U (le *until* fort) via l'équivalence

$$\varphi \mathsf{W} \psi \;\equiv\; (\varphi \mathsf{U} \psi) \vee \mathsf{G}\varphi$$

La propriété souhaitée est alors soumise à SMV, et prouvée, sous la forme

$$\mathsf{G}\,(\mathtt{MG} \rightarrow ((\mathsf{G}\,\neg\mathtt{MD}) \vee (\neg\mathtt{MD}\;\mathsf{U}\;(\neg\mathtt{MG} \wedge \neg\mathtt{MD}))))$$

IV.2.4 Conservation du sens de marche:

Nous voulons maintenant vérifier que, si le bouton go est appuyé mais pas le bouton ann_req, alors le moteur se met en marche jusqu'à ce qu'on appuie sur le bouton stop ou que la mission arrive à son terme (ce qui peut se produire immédiatement). La formule LTL correspondante est analogue à la précédente :

$$\begin{aligned}&\mathsf{G}\,((\mathtt{go} \wedge \neg\mathtt{ann\_req})\\ &\rightarrow ((\mathtt{MG} \vee \mathtt{MD})\;\mathsf{W}\;(\mathtt{stop} \vee (\mathtt{pos} = \mathtt{dest}))))\end{aligned}$$

Nous effectuons la traduction du W en U, mais, de même qu'en IV.2.2, la propriété ne peut pas être vérifiée directement. Nous transformons alors la formule en une autre formule prenant en compte la synchronisation sur la fin de cycle. La nouvelle formule correspond bien à la propriété attendue, et est prouvée par SMV.

IV.2.5 Prise en compte d'hypothèses comportementales:

La formule précédente utilise le connecteur W parce que nous n'avons fait aucune hypothèse sur l'évolution des variables d'entrée. En particulier, nous ne supposons pas que le chariot arrivera forcément un jour à destination, ou que la requête sera annulée. Afin de refléter le comportement nominal de la totalité ou d'un sous-ensemble de la partie opérative, on fait des hypothèses sur l'évolution des variables d'entrée. Ces hypothèses reposent sur la connaissance de lapartie opérative et doivent être explicitement introduites dans la formalisation des propriétés attendues.

Les propriétés attendues ont alors la forme générale

$$hyp. \rightarrow prop.$$

qui se lit « le long de toutes les exécutions satisfaisant *hyp.* la propriété *prop.* est vérifiée ». On parle d'*hypothèses dynamiques* lorsque des combinateurs temporels sont utilisés, comme par exemple dans l'hypothèse

$$\mathsf{G}\,((\mathtt{MG} \vee \mathtt{MD}) \rightarrow \mathsf{F}\,(\mathtt{stop} \vee (\mathtt{pos} = \mathtt{dest}))$$

qui se lit « si le moteur est mis en marche, un des deux événements suivants finit par se produire : le bouton stop est appuyé ou le chariot arrive à destination ».

Nous pouvons utiliser cette hypothèse pour prouver que le chariot conserve son mode de marche jusqu'à l'arrêt, et qu'il ne se déplace pas indéfiniment. La propriété *prop.* est alors :

$$\begin{aligned}&\mathsf{G}\,((\mathtt{go} \wedge \neg \mathtt{ann\_req})\\ &\rightarrow ((\mathtt{MG} \vee \mathtt{MD})\,\mathsf{U}\,(\mathtt{stop} \vee (\mathtt{pos} = \mathtt{dest}))))\end{aligned}$$

SMV nous indique alors que $hyp. \rightarrow prop.$ est satisfaite par notre système.

V Conclusion

Dans ce papier nous avons présenté un cadre formel permettant la vérification automatique de programmes IL simples. Les éléments constitutifs de notre approche sont :

- le choix d'un sous-ensemble pertinent du langage IL ;
- une sémantique opérationnelle précise et claire du langage retenu, à base de systèmes de transitions ;
- un codage de ces systèmes de transitions dans le langage d'entrée d'un model-checker tel que SMV ;
- l'utilisation de la logique temporelle LTL pour l'expression des propriétés dynamiques à vérifier.

Cette approche, fondée sur des techniques éprouvées dans le domaine de l'informatique, permet de prouver ou d'invalider, d'une façon complètement automatique, la correction de programmes IL de taille non triviale.

Dans le cadre de nos travaux sur la validation de programmes API, une étude similaire a déjà été menée sur le langage LD [13]. À terme, notre but est de proposer aux concepteurs de programmes API un ensemble de méthodes simples et efficaces permettant d'effectuer la validation de programmes. Nous travaillons actuellement à l'extension de la méthode proposée dans cet article à un sous-ensemble élargi du langage IL et aux programmes multilangages.

RÉFÉRENCES

[1] CEI (Commission électrotechnique internationale). *CEI 61131-3 : Automates programmables - partie 3*, 1993.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[3] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science Publishers, 1990.

[4] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behavior by means of Timed Net Condition/Event Systems. In *Proc. 6th IEEE Conf. Emerging Technologies and Factory Automation (ETFA'97), Los Angeles*, pages 110–122, 1997.

[5] M. Heiner and T. Menzel. Instruction list verification using a Petri Net semantics. In *IEEE Int. Conf. on Systems, Man and Cybernetics, San Diego*, pages 716–721, 1998.

[6] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Proc. 4th IEEE Workshop on Discrete Event Systems (WODES'98), Cagliari, Italy*, pages 161–165, 1998.

[7] F. Jimenéz-Fraustro and É. Rutten. Modélisation synchrone de standards de programmation de systèmes de contrôle : le langage ST de la norme CEI 1131-3. In *Journée d'études « Nouvelles percées dans les langages pour l'automatique », SEE – Club 18 (Automatique et Automatisation Industrielle), Amiens (France)*, 1999.

[8] S. Kowalewski, S. Engell, and O. Stursberg. Verification of logic controllers for continuous plants. In *Advances in Control : Highlights of ECC'99*, pages 345–389. Springer-Verlag, 1999.

[9] 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*, 1999.

[10] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proc. 11th Euromicro Conference on Real-Time Systems (ECRTS'99), York, UK*, 1999.

[11] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs. http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps.

[12] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[13] 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*, 2000.

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

[15] N. Völker and B.J. Krämer. Modular verification of function block based industrial control systems. In *Proc. 24th IFAC/IFIP Workshop on Real-Time Programming*, 1999.