Vérification automatique de programmes écrits dans les langages IL et ST de la norme IEC 61131-3

Thèse de
Géraud Canet

17 décembre 2001

Table des matières

1. Introduction

1.1. Vérification des logiciels

1.1.1. Conception sûre des logiciels

Lorsqu'on parle de la partie logicielle d'un appareil, il n'est pas rare de s'entendre évoquer les inévitables erreurs de programmation (les fameux *bugs*), et les tristes pannes qu'elles infligeront au système et à son utilisateur. Influencés par l'expérience des systèmes informatiques que nous utilisons au quotidien, autant que par le souvenir de quelques pannes spectaculaires (nous ne citerons que le légendaire premier lancer d'Ariane 5, mais cet exemple, bien que rare à une telle échelle, n'est pas unique), nous en arrivons à envisager la panne comme une conséquence indissociable de la programmation. C'est oublier que les logiciels n'équipent pas que des micro-ordinateurs, mais surtout des automates de production dans des usines, des automobiles, des avions : autant de systèmes pour lesquels la panne n'est, en principe, pas autorisée. Que ce soit pour des raisons humaines (les exemples précédemment cités, mais encore la régulation du trafic routier ou ferroviaire, une usine chimique...), ou pour des raisons financières (erreur provoquant l'arrêt prolongé d'une usine, *bug* diffusé en plusieurs dizaines de milliers d'exemplaires dans des automobiles en circulation...), ces logiciels, dits critiques, ne doivent pas être diffusés si leur utilisation comporte un risque. Leur sûreté doit être, autant que possible, garantie.

Concevoir un logiciel sûr mène à s'interroger sur sa validation. Celle-ci est généralement effectuée, tout au long du processus de conception, au moyen de différentes méthodes. Il existe un large choix de méthodes permettant la validation des logiciels ; la décision de mettre en œuvre l'une ou l'autre de ces méthodes dépend directement de la criticité du projet, et joue un rôle sur le processus de conception, la compétence requise des ingénieurs, et naturellement sur le coût de fabrication.

1.1.2. Validation par la simulation non formelle

Dans le cas général, la validation d'un logiciel, ou d'un programme d'automate industriel, est effectuée en le faisant tourner. Le résultat est comparé à la spécification du produit, et à son cahier des charges. Ceux-ci sont généralement des documents rédigés, présentés de manière non formelle (un langage mathématique est rarement utilisé). L'opération suppose la mobilisation d'une équipe de testeurs (concepteurs et utilisateurs). Comme souligné par [FL00], ce type de validation a deux défauts : elle n'est jamais terminée, et elle est fort coûteuse en temps et en argent. De plus, cette approche ne permet de garantir la validité du programme que pour certaines exécutions (il existe naturellement un lien entre le budget alloué à la vérification et le taux de couverture qui en résulte). Cette méthode ne permet pas de garantir qu'une propriété donnée est vérifiée dans tous les cas.

La validation prend d'autant plus d'importance dans le cas des systèmes critiques. La méthode de validation traditionnelle, trop laborieuse en fonction des objectifs à atteindre, ne suffit plus. De nouvelles techniques, utilisant des méthodes dites « formelles » (car elles reposent sur l'utilisation de modèles mathématiques), permettent de mettre en place d'autres formes de validation, qui peuvent être partiellement automatisées, et mises en œuvre très tôt dans la conception du système.

1.1.3. Méthodes formelles

Les méthodes formelles, par l'utilisation de modèles et de techniques reposant sur des définitions mathématiques, offrent de nouvelles possibilités pour la validation des logiciels. Leurs mise en application permet de garantir des propriétés fondamentales du système, avec pour résultat concret d'éliminer plus d'erreurs. Les techniques employées sont relativement simples et peu coûteuses à mettre en œuvre, et sont d'autant plus rentables qu'elles permettent de déceler des fautes de conception au plus tôt. Leur utilisation exige cependant la compétence d'un ingénieur qualifié. Il y a donc, en général, un compromis à trouver entre l'efficacité de la validation, la compétence du personnel chargé de l'effectuer, et le coût de sa mise en place.

La validation formelle d'un système suppose la définition d'un modèle de ce système. Une telle définition est délicate et elle est souvent réalisée par des experts. Cependant, la définition d'un modèle générique à une classe de problèmes, ou de langages, permet ensuite de pouvoir établir, de manière automatique, un modèle adapté à toute situation appartenant à la même classe. Le travail exposé dans cette thèse présente une telle modélisation, adaptée aux langages de programmation des automates programmables industriels. Cette modélisation faite, il est possible de développer un compilateur traduisant automatiquement tout programme écrit dans les langages traités vers le modèle qui lui est associé.

Parmi les méthodes existantes, afin de définir des méthodes formelles pour la validation de programmes d'automates industriels, nous avons été amenés à nous intéresser à la technique du model-checking symbolique. Cette technique prometteuse nous semble offrir un très bon compromis entre la couverture des différentes exécutions du système, l'efficacité de la validation, le raisonnable niveau de compétence requis pour son utilisation relativement à d'autres méthodes, et la simplicité de sa mise en œuvre.

Model-checking Étant donnée une propriété, le problème du model-checking consiste à établir si cette propriété est vérifiée ou non par toutes les exécutions possibles du système. Cette méthode a l'avantage de l'exhaustivité, en offrant, pour une propriété donnée, la couverture totale des exécutions possibles. La validation dépend donc essentiellement des formules que l'on vérifie (il est toujours possible d'oublier de vérifier une formule qui aurait permis de trouver de nouveaux *bugs*).

Une grande part du processus étant automatisable, les compétences requises pour la pratique de la vérification par model-checking sont minimes si les propriétés à vérifier sont déjà établies ou très simples. En revanche, si les propriétés à vérifier sont plus riches, variées ou complexes, il sera nécessaire d'avoir la connaissance d'un formalisme d'écriture des propriétés (par exemple une logique temporelle, voir la section 5.2 page 92). Le model-checking est alors une méthode puissante de validation, dans les cas où il est réalisable.

Model-checking symbolique La principale difficulté posée par le model-checking est que l'exploration exhaustive des exécutions du système a pour conséquence une très forte complexité. Cependant, la vérification est souvent rendue possible, grâce à des méthodes d'optimisation largement utilisées. En particulier, la technique du model-checking symbolique, que nous exploitons pour nos travaux, permet, par des méthodes puissantes de représentation (qui évitent de devoir représenter en mémoire les états du système), d'effectuer la vérification de systèmes de très grande taille.

Les principes du model-checking symbolique, et son utilisation (notamment dans le cadre de ce projet), sont décrits en détail au chapitre 5.

1.2. Automates programmables industriels (API)

Les automates programmables industriels (API)[1] sont des appareils de contrôle-commande utilisés pour des systèmes automatisés (usines industrielles, jouets, ascenseurs, électro-ménager, trafic routier ou ferroviaire...). Couramment utilisés dans des domaines critiques, la validation des programmes pour API est d'une importance primordiale. Nous énonçons ci-après les principales particularités des API, puis nous présentons les langages traditionnels servant à leur programmation ; dans la suite, nous nous concentrons sur les langages de la norme IEC 61131-3, et en particulier IL et ST, qui font l'objet du travail présenté dans cette thèse.

1.2.1. Éléments caractéristiques du comportement des API

1.2.1.1. Gestion des entrées-sorties

L'automate programmable est un instrument de calcul et de commande qui sert d'interface à des capteurs, appareils, robots, etc. auxquels il est relié physiquement. Pour assurer la communication avec ces appareils, l'automate distingue les signaux d'entrée, récupérés par les *capteurs*, des signaux de sortie, transmis par les *actionneurs*. Dans un programme, il existe donc des variables spéciales permettant de représenter les entrées et les sorties. Les variables d'entrée sont généralement accessibles en lecture seule, et les variables de sortie en écriture seule. Ces variables, comme nous le précisons à la section suivante, sont mises à jour par l'automate programmable selon une procédure précise.

[1] *PLC* en anglais (*Programmable Logic Controller*).

1.2.1.2. Comportement cyclique

Les automates industriels que nous étudions fonctionnent de manière cyclique. Un cycle peut être décomposé en trois temps :
- acquisition des entrées ;
- traitements internes (dont exécution du programme) ;
- affectation des sorties.

Dans notre étude, les traitements internes se réduisent à l'exécution du programme. Après chaque exécution d'un cycle, un nouveau cycle est immédiatement relancé. Ce mode de fonctionnement fait l'originalité du comportement des API ; nous en tiendrons compte lorsque nous serons amenés à définir un modèle de leur comportement (chapitre 4).

1.2.2. Programmation des API

La programmation d'un API s'effectue à l'aide de langages spécialisés, fournis par son constructeur. Les langages couramment utilisés pour la programmation des API appartiennent en général à l'une ou l'autre des grandes familles suivantes :
- langage machine : il s'agit d'une suite d'instructions très simples, impliquant généralement des opérations mémoire primaires (à base d'un tableau de registres) ;
- grafcet : variante spécialisée des réseaux de Petri, le grafcet est un langage graphique fréquemment utilisé pour la spécification des systèmes automatisés. Il consiste en un ensemble d'étapes reliées entre elles par des transitions. Chaque étape peut être active ou inactive ; les règles qui régissent l'activation et la désactivation des étapes sont fondées sur une évaluation des transitions en fonction du contexte courant (valeur des variables notamment). Le grafcet offre l'avantage de permettre une spécification synthétique et modulaire, sous forme graphique, d'un système automatisé complexe ;
- schéma à relais (*ladder diagram*) : ce langage est fondé sur une analogie entre les flux de données d'un programme et le courant électrique dans un circuit série-parallèle. Un programme est un diagramme en forme d'échelle ; chaque échelon est « porteur » d'une valeur booléenne, et se voit affecter un calcul sur des valeurs de variables, représentées par des *contacts* reliés entre eux par des liaisons série-parallèle. Au bout de chaque échelon figure une *bobine* représentant l'affectation du résultat dans une variable. À l'origine, les programmes d'automates industriels étaient directement câblés par des liaisons électriques. Le schéma à relais était le langage naturel de tels systèmes, et son usage est resté par tradition ;
- logigrammes : il s'agit d'une représentation du flux de données du programme à la manière d'un circuit électronique (à l'aide de portes *ou*, *non*, etc.).

Les trois derniers langages cités sont des langages graphiques (un programme est défini par un graphe, alors que dans l'usage informatique courant il est défini par un texte). Parmi ces langages, le grafcet tient une place un peu particulière, étant utilisé, par certains environnements, aussi bien pour la spécification que pour la programmation.

Ces langages (dont la liste n'est pas limitative) doivent être considérés comme des familles auxquelles se rattachent les langages fournis par les différents constructeurs.

1.3. La norme IEC 61131-3

1.3.1. Motivation

Pour la programmation des API qu'il produit, chaque constructeur propose un ou plusieurs langages, utilisable au travers d'une interface (qui peut être un pupitre, ou un logiciel sur PC). Les langages appartiennent en général à l'une ou l'autre des familles citées précédemment. Cependant, bien que semblables et de la même famille, les langages de deux constructeurs différents ne sont pas forcément compatibles. Cela rend évidemment délicate la portabilité logicielle des programmes d'API, en spécialisant excessivement un programme au matériel qui le supporte. La norme IEC 61131-3 résulte de la volonté des constructeurs et des utilisateurs de travailler sur des langages normalisés.

En même temps que la norme a été rédigée, un comité nommé PLC-Open[2] a été créé, dont la mission est de promouvoir la norme et de gérer ses évolutions.

1.3.2. Les cinq langages de la norme

La norme IEC 61131-3 définit cinq langages dédiés à la programmation des API :

SFC (*Sequential Function Chart*) Ce langage appartient à la famille du grafcet, avec lequel il comporte quelques différences, tendant vers la simplification et la rigueur. Il peut être utilisé aussi bien pour la spécification que pour la programmation ;

IL (*Instruction List*) Il appartient à la famille des langages machine. Sa définition dans le cadre de ce projet est décrite en détail à la section 2.3 page 28 ;

ST (*Structured Text*) Il s'agit d'un langage textuel structuré, de type Pascal ou C. Il est décrit en détail à la section 2.4 page 33 ;

LD Membre de la famille des *Ladder Diagrams* ;

FBD (*Function Block Diagram*) FBD est un langage de représentation des « blocs fonctionnels » du programme. Il permet de considérer un programme de manière globale et modulaire, en faisant l'abstraction du code interne de chaque bloc fonctionnel (approche « boîte noire »). Les emboîtements sont représentés par des liens unissant entre elles les variables d'interface des blocs fonctionnels.

La norme définit de plus une représentation textuelle des langages SFC et FBD. Cette représentation permet de « taper » un programme SFC ou FBD sans disposer d'éditeur graphique, avec un compilateur normal.

Les cinq langages permettent une large couverture des usages et traditions en vigueur dans la communauté des programmeurs d'API. Ajoutons que les langages ont été conçus de manière à pouvoir être combinés entre eux, de manière modulaire (nous désignerons de tels programmes sous le nom de programmes multilangages). Le SFC est alors utilisé comme langage structurant, c'est-à-dire celui dans lequel est écrit le programme principal.

[2] `www.plcopen.org`.

1.3.3. Discussion autour de la norme

Dans le but de faciliter la réutilisation logicielle et l'interchangeabilité matérielle des API, l'apport de la norme est considérable. Mais, comme on l'imagine, la définition des langages est le fruit d'une longue négociation entre constructeurs et développeurs, dont il résulte inévitablement une série de compromis sur lesquels la norme est construite. La norme est souvent peu claire, certaines définitions sont manquantes ou ambiguës. La norme stipule par exemple qu'il doit être possible de traduire n'importe quel langage dans n'importe quel autre, mais ne donne pas d'algorithme, ni d'indice sur la traduction (qui est loin d'être triviale). La résolution des ambiguïtés est indispensable à l'étude d'un langage. Que ce soit pour définir une méthode de validation ou pour programmer un compilateur, un langage clair doit au préalable avoir été défini. Dans certains cas, cette définition à elle seule justifie des travaux particuliers, comme par exemple [Tou97] dans le cas des FBD.

Nous donnons au chapitre 2 une définition non ambiguë des langages IL et ST. Pour nous orienter dans nos choix, une de nos principales sources était l'ouvrage de Lewis [Lew98], manuel de référence qui propose une relecture pédagogique de la norme, en ajoutant le cas échéant des précisions.

1.3.4. Outils

Des outils commerciaux supportant partiellement la norme IEC 61131-3 existent sur le marché ; ils permettent de spécifier un logiciel de commande et d'en générer le code, mais ne permettent actuellement pas de valider le code par rapport au cahier des charges. Dans le cadre de notre travail, nous avons eu accès à l'utilisation du logiciel ISaGRAF ([CJ 98]), dont l'utilisation est très répandue dans le domaine des automates industriels. Le logiciel ISaGRAF est un simulateur/compilateur de programmes d'API, permettant d'extraire automatiquement un code exécutable sur le modèle particulier d'automate dont on dispose (pour peu qu'un pilote dédié ait été programmé). Vu que le logiciel permet également d'effectuer la simulation de programmes, nous avons été amenés à l'utiliser, à des fins d'expérimentation, au cours du projet. En particulier, lorsque le cas d'une forme de langage incomplètement ou non spécifiée par la norme se présentait, les choix effectués par les concepteurs d'ISaGRAF ont été pris en considération. Les choix implémentés sous ISaGRAF n'ont cependant pas été systématiquement adoptés lors du projet Vulcain : les décisions ont été prises en tenant compte de la littérature, et de l'expérience des membres automaticiens du projet.

1.4. Projet Vulcain

1.4.1. Buts du projets

Le projet Vulcain [dCR$^+$00] s'est fixé pour but la validation des programmes d'API. L'objectif est de définir une méthode complète permettant de valider un programme d'API utilisant un langage normalisé.

Les langages choisis sont les langages SFC, LD et ST de la norme IEC 61131-3. Ces trois langages représentent des habitudes de programmation différentes, et permettent une bonne couverture des programmes d'API. Il est à noter que les méthodes définies dans le cadre du projet permettent la validation d'un programme multilangage, c'est-à-dire écrit dans n'importe quelle combinaison valide des trois langages étudiés.

1.4.2. Acteurs

Le projet était un projet industriel et universitaire, dans lequel étaient engagés :

- Alcatel R&I (*Research and Innovation*, Marcoussis), qui avait notamment pour charge de fournir des cas d'étude, et des exemples de travail ;
- le laboratoire LSV (ENS de Cachan, CNRS UMR 8643), pour son expertise dans la vérification formelle des programmes et des systèmes automatisés ;
- le laboratoire Lurpa (ENS de Cachan), pour son expertise dans le domaine de l'automatique et des langages de la norme.

Le projet a impliqué, à temps complet ou partiel, plusieurs ingénieurs d'Alcatel et enseignants-chercheurs de l'ENS de Cachan, une post-doctorante et deux doctorants. Le rapport final du projet a été rendu en février 2001.

1.4.3. Originalité du travail

La principale originalité du projet était de fournir des méthodes de vérification pour des programmes déjà existants. En effet, on constate que les travaux autour de la validation de programmes (on trouvera au chapitre 3 un état de l'art des modèles développés autour des langages de la norme), dans le domaine de l'automatique, se traduisent souvent par des recommandations ou des actions d'ordre méthodologique. Au contraire, les résultats du projet Vulcain n'impliquent pas de contrainte particulière sur le programme (hormis le fait qu'il soit valide syntaxiquement, et ne comporte pas d'erreur flagrante de compilation[3]), et permettent la prise en compte de tout programme valide tel quel.

La méthode proposée par le projet Vulcain est, de plus, complète, dans la mesure où elle couvre tous les aspects de la validation formelle, de la définition du modèle jusqu'à la vérification de propriétés. Une fois les définitions effectuées, le processus est presque entièrement automatique. Seule la formalisation des propriétés requiert une intervention, décrite dans ce document.

Enfin, la prise en compte de programmes multilangages est une réelle originalité du projet, qui définit pour cela une sémantique des trois langages dans un formalisme unifié, puis une sémantique de leur combinaison.

[3]Une telle erreur serait par exemple l'utilisation d'une variable non déclarée, ou la double déclaration d'une variable, etc.

1.4.4. Impact de la thèse dans le projet Vulcain

Le travail de thèse qui fait l'objet de ce rapport a directement contribué aux résultats du projet Vulcain, par la définition complète de la sémantique des langages IL et ST, et l'intégration de cette dernière à la sémantique multilangage. La prise en compte du langage IL n'était pas contractuelle dans le projet Vulcain, mais elle nous a semblé utile, en raison notamment de la plus vaste utilisation des langages de type IL dans le monde automatique, par rapport à ST. De plus, l'étude de IL est proche, en plus simple, de celle de ST, et permettait rapidement d'en effectuer la faisabilité.

Le codage de ces sémantiques dans le langage du model-checker SMV a également contribué au projet. De manière globale, même si la sémantique multilangage n'est pas décrite dans ce document, notre participation à l'ensemble du processus a permis l'émergence d'une telle sémantique, qui a pu être utilisée pour la vérification de systèmes réels.

1.5. Contenu de la thèse

1.5.1. Description de notre approche

Notre approche consiste à définir une méthode complète permettant la validation, par la technique du model-checking symbolique, de programmes écrits dans les langages IL et ST de la norme IEC 61131-3. Cette méthode comprend les étapes suivantes :

1. définition précise et formelle des fragments de langage étudiés ;
2. définition d'une sémantique opérationnelle de ces langages. Cette sémantique opérationnelle définit le modèle de chaque programme valide ;
3. définition d'un codage des modèles de programmes, à partir de la sémantique opérationnelle, vers le langage d'un model-checker ;
4. expérimentation sur des exemples.

Chacune des étapes utilise uniquement les résultats de la précédente. Par exemple, pour définir le codage d'un programme, il n'est pas nécessaire de disposer du programme lui-même. Le codage est uniquement défini à partir du modèle du programme (sa sémantique opérationnelle). Lors de la mise en pratique (codage et expérimentation), c'est le logiciel Cadence SMV (model-checker symbolique [McM]) qui a été retenu.

1.5.2. Plan de la thèse

Dans un premier temps, nous présentons au chapitre 2 la définition précise des langages étudiés. Nous y donnons la définition formelle d'un fragment booléen de chaque langage, qui sera utilisé dans la suite des travaux.

Nous présentons au chapitre 3 un état de l'art des modélisations des langages de la norme (en particulier IL et ST). Notre modélisation, quant à elle, est présentée au chapitre 4, par la définition de la sémantique opérationnelle des langages. Cette définition s'applique aux fragments de langages définis précédemment.

Afin d'introduire la vérification formelle par la technique du model-checking symbolique, nous en présentons les principes et la mise en application au chapitre 5. On y trouvera notamment :

- une présentation des logiques temporelles utilisées pour exprimer les propriétés, ainsi que des exemples de propriétés typiques ;
- une description des principes et particularités du model-checking symbolique ;
- une présentation du logiciel Cadence SMV, de sa syntaxe et son utilisation. On y indiquera également, en les motivant, quelques recommandations d'usage utiles à connaître.

Le logiciel SMV sera directement exploité au chapitre 6, dans lequel nous définissons le codage de la sémantique opérationnelle IL et ST vers le format d'entrée de SMV. Enfin, au chapitre 7, nous illustrons notre approche par son expérimentation sur deux exemples.

2. IL et ST, les langages textuels de la norme

2.1. Introduction

2.1.1. Langages traités

Nous nous intéressons aux deux langages textuels de la norme, IL et ST. Le langage IL est un langage de type assembleur, composé de lignes de codes ; le langage ST est un langage structuré de type Pascal, ADA ou C, composé de termes. Pour la description des langages, nous nous appuierons sur la norme, en version originale [IEC93] et française [CEI93], sur le manuel de R. W. Lewis [Lew98] et sur le manuel du logiciel ISaGRAF [CJ 98].

Nous considérons ici une version simplifiée des langages IL et ST, desquels nous retirons certaines notions. Par exemple, les traitements des dates, ou des structures de données composées (tableaux, chaînes, structures), ne sont pas pris en compte ; on suppose que tout traitement se ramène à un traitement sur les booléens et les entiers bornés, et que les programmes se limitent à un seul bloc fonctionnel (pas d'appel de sous-programme). De même, nous simplifions la définition des éléments lexicaux (identificateurs, entiers...), qui comportent des subtilités inutiles ici[1]. La définition des deux langages que nous retenons est plus simple que celle de la norme, mais elle ne modifie pas fondamentalement leur pouvoir d'expression. Les langages que nous considérons sont entièrement décrits dans ce chapitre.

De ces langages, nous définissons formellement des fragments booléens, que nous appelons IL_0 et ST_0. C'est sur ces langages que sera définie la sémantique opérationnelle.

L'annexe A page 171 récapitule la grammaire correspondant au fragment des langages que nous prenons en considération. Cette grammaire est un sous-ensemble de celle donnée par la norme.

[1] Par exemple, selon la norme, l'entier `2_345` est valide et est équivalent à `2345`, mais `_2_345` et `2__345` sont invalides. Le caractère de soulignement a dans ce cas une valeur purement décorative, puisqu'il ne modifie pas le nombre.

2.1.2. Contenu du chapitre

Ce chapitre contient une définition de chacun des deux langages. Cette définition présente les aspects syntaxiques du langage et une description de son interprétation, c'est-à-dire une sémantique décrite informellement[2]. Ce chapitre décrit les éléments communs aux langages de la norme, puis donne, pour chacun des deux langages textuels :

1. une présentation du langage, décrivant sa philosophie générale et son utilisation, suivie d'une description plus détaillée des éléments qui le composent. Nous y donnons une description informelle du comportement de chacun des éléments ;
2. la grammaire concrète du fragment étudié, au format utilisé par la norme. C'est la référence de la syntaxe du langage, elle peut être utilisée pour en programmer un analyseur syntaxique ;
3. une définition des programmes dits « bien formés », c'est-à-dire non seulement syntaxiquement corrects, mais respectant également un ensemble de règles nécessaires pour qu'une sémantique puisse être définie (par exemple, un saut vers une étiquette exige que l'étiquette ait été déclarée quelque part) ;
4. une définition de la syntaxe abstraite du langage. Il ne s'agit pas à proprement parler d'une grammaire dans le sens de l'analyse syntaxique, mais plutôt de donner à chaque forme de langage une structure mathématique, que nous utiliserons ensuite pour en définir la sémantique.

La syntaxe abstraite permet de donner une définition formelle des langages. Pour chacun des langages, nous donnons une syntaxe abstraite d'un fragment booléen, comme le décrit la section suivante.

2.1.3. Fragments booléens définis formellement : les langages IL_0 et ST_0

Nous donnons une définition formelle (une syntaxe abstraite) pour des fragments booléens des langages IL et ST, que nous appelons IL_0 et ST_0. Ce sont les fragments pour lesquels une sémantique opérationnelle est directement définie au chapitre 4. Nous serons amenés, par la suite, à présenter le cas échéant des méthodes pour la prise en compte des entiers bornés.

Il n'est en général pas nécessaire de définir directement une sémantique opérationnelle pour tous les éléments du langage. En effet, de nombreux éléments de langage sont définis dans un but pratique (faciliter l'écriture ou la lisibilité des programmes), mais n'ajoutent rien au pouvoir d'expression du langage. Par exemple, une boucle `REPEAT` ... `UNTIL` peut être récrite à l'aide d'une boucle `WHILE` ... `DO` : la différence qui découle de la récriture, en termes de complexité, est négligeable. Nous ne définirons donc pas directement de sémantique opérationnelle pour les formes de langage qui peuvent être récrites à partir d'autres formes pour lesquelles une sémantique est déjà définie.

[2] Une sémantique formelle sera ensuite proposée, pour chaque langage, au chapitre suivant.

type	borne inférieure	borne supérieure
`SINT`	-2^7	2^7-1
`INT`	-2^{15}	$2^{15}-1$
`DINT`	-2^{31}	$2^{31}-1$
`LINT`	-2^{63}	$2^{63}-1$
`USINT`	0	2^8
`UINT`	0	2^{16}
`UDINT`	0	2^{32}
`ULINT`	0	2^{64}

TAB. 2.1.: Intervalle de définition des types entiers de base.

2.2. Éléments communs aux deux langages

2.2.1. Éléments lexicaux

2.2.1.1. Constantes

Nous désignons les constantes booléennes par les mots-clés `TRUE` et `FALSE`[3]. Les entiers sont une suite de chiffres. Les entiers signés sont des entiers, éventuellement précédés d'un signe `+` ou `-`.

2.2.1.2. Identificateurs

Les identificateurs sont une suite de caractères alphabétiques pouvant contenir des caractères de soulignement et des chiffres, sauf en première position. De plus, deux caractères de soulignement ne peuvent pas se trouver côte à côte. Par exemple, `A01`, `bn` et `A_X` sont des identificateurs valides ; `1X` et `X__A` ne sont pas des identificateurs valides. Naturellement, les mots-clés du langage ne sont pas des identificateurs valides.

2.2.2. Types de données

2.2.2.1. Types de bases

Nous considérons deux types de données : les booléens et les entiers bornés.

Le type de base pour les booléens est noté `BOOL`. Pour les entiers, la norme fournit huit types de base et fixe leurs intervalles de définition (tableau 2.1).

Valeurs initiales par défaut Selon la norme, tout type possède une valeur initiale par défaut. Pour le type `BOOL`, cette valeur est `FALSE` ; pour tous les types entiers de base, cette valeur est `0`.

[3]La norme propose de nombreuses manières de noter les booléens ; nous nous tiendrons à celle-ci.

2.2.2.2. Entiers sous-spécifiés

La norme permet de spécifier explicitement l'intervalle de définition de certains types d'entiers, dits entiers « sous-spécifiés », car créés à partir d'un type standard dont on réduit l'intervalle de définition. Par exemple, l'expression `INT (0..5)` désigne le type `INT` dont l'intervalle de définition est réduit à [0,5][4].

Il est souvent judicieux pour un programmeur de définir strictement les entiers à leur intervalle utile : cela permet, d'une part, d'optimiser la taille mémoire occupée par le processus, et, d'autre part, de bénéficier le cas échéant d'un contrôle de dépassement des bornes. Dans le cas de la vérification par model-checking, où la taille mémoire occupée par les variables joue un rôle primordial, il est très utile de pouvoir régler l'intervalle de définition d'un entier.

Afin d'exploiter la fonctionnalité des entiers sous-spécifiés pour simplifier le traitement des entiers (par exemple pour diminuer la complexité de la vérification), nous considérons que l'expression `INT (0..5)` désigne un type d'entier dont l'intervalle de définition est [0,5]. Il en découle le fait que, par exemple, `INT (0..5)` est équivalent à `USINT (0..5)` (le type de base n'importe pas).

Valeur initiale par défaut La valeur initiale par défaut d'un type sous-spécifié est la borne inférieure de son intervalle de définition. Par exemple, pour le type `INT (-1..3)`, la valeur initiale par défaut est `-1`.

2.2.2.3. Identifiants de types

Un identifiant de type est un symbole, ou une suite de symboles permettant de désigner un type de données. Il peut s'agir :
- d'un mot-clé désignant un des types de base (voir la section 2.2.2.1) ;
- d'un identifiant de type entier sous-spécifié (voir la section 2.2.2.2) ;
- d'un identificateur.

Dans le dernier cas, il s'agit d'un type « dérivé » (voir la section suivante).

2.2.2.4. Types dérivés

Un type dérivé est un type défini par l'utilisateur. Il comporte les caractéristiques suivantes :
- un identificateur (le *nom* du type) ;
- un identifiant de type (décrit le type *parent*) ;
- une valeur initiale par défaut.

[4]En fait, bien que cela ne soit pas précisé explicitement par la norme, nous supposons que le type de base permet de spécifier la place occupée par l'entier en mémoire, et l'intervalle précise les valeurs permises pour les entiers de ce type : `INT (0..5)` est alors un entier sur 16 bits dont seules les valeurs de 0 à 5 sont permises.

```
TYPE
  a : BOOL := TRUE;
  b : INT (-1..3);
  c : b := 0;
END_TYPE
```

FIG. 2.1.: Exemple de définition de types dérivés.

Définition par l'utilisateur d'un type dérivé Dans un programme, on définit un type dérivé au moyen d'un bloc `TYPE` ... `END_TYPE`. Il n'est pas obligatoire de mentionner explicitement la valeur initiale par défaut ; dans ce cas, la valeur est celle du type parent. Par exemple, la figure 2.1 montre une définition de trois types personnalisés `a`, `b` et `c` tels que :
- `a` est un type de booléens dont la valeur initiale par défaut est `TRUE` ;
- `b` est un type d'entiers dont l'intervalle de définition est [`-1`,`3`] et dont la valeur initiale par défaut est `-1` (valeur par défaut du type parent, voir la section 2.2.2.2) ;
- `c` est un type d'entiers dont l'intervalle de définition est [`-1`,`3`] et dont la valeur initiale par défaut est `0`.

2.2.3. Variables

Une variable comporte les caractéristiques suivantes :
- un identificateur (le nom de la variable) ;
- un identifiant de type (le type de la variable) ;
- une valeur initiale.

Dans les langages de la norme, toute variable doit posséder une valeur initiale. Si celle-ci n'est pas précisée à la déclaration de la variable, on utilise la valeur initiale par défaut de son type.

Entrées et sorties On distingue quatre types de variables, en fonction de leur rôle dans la gestion des entrées-sorties (voir la section 1.2.1.1 page 13) :
- les variables d'entrée, qui correspondent aux valeurs reportées par les capteurs. Elles ne sont pas accessibles en écriture ;
- les variables de sortie, qui correspondent aux valeurs à transmettre aux actionneurs. Elles ne sont pas accessibles en lecture ;
- les variables d'entrée-sortie, accessibles aussi bien en lecture qu'en écriture ;
- les variables internes, qui sont utilisées pour les calculs.

Déclaration des variables Les variables sont déclarées respectivement au moyen des blocs suivants : `VAR_INPUT` ... `END_VAR`, `VAR_OUTPUT` ... `END_VAR`, `VAR_IN_OUT` ... `END_VAR`, `VAR` ... `END_VAR`. Il n'est pas obligatoire de mentionner explicitement la valeur initiale par défaut ; dans ce cas, la valeur est celle du type de la variable. Par exemple, la figure 2.2 page suivante montre la déclaration de trois variables `A`, `B` et `C` telles que :

```
VAR
  A : BOOL ;
  B : INT := 5 ;
  C : b := 2 ;
END_VAR
```

FIG. 2.2.: Exemple de déclaration de variables.

Fonction	Opérateur ST	Description
ADD	+	addition
SUB	-	soustraction
MUL	*	multiplication
EXPT	**	élévation à une puissance
DIV	/	division entière
MOD		modulo

TAB. 2.2.: Fonctions arithmétiques standard.

- `A` est une variable booléenne dont la valeur initiale est `FALSE` (valeur par défaut des booléens, voir la section 2.2.2.1) ;
- `B` est une variable de type `INT` mais de valeur initiale `5` ;
- `C` est une variable entière bornée à l'intervalle [`-1`, `3`] (type `b`) et sa valeur initiale est `2`).

2.2.4. Fonctions

Les fonctions standard sont définies par la norme, indépendamment des langages. Nous présentons ici les fonctions arithmétiques, booléennes et les fonctions de comparaison. Nous ne présentons pas les fonctions sur les réels (`LN`, `SIN`...), que nous ne prenons pas en compte, ni les fonctions de sélection (`SEL`, `MIN`, `MAX`, `LIMIT`, `MUX`), qui peuvent facilement être récrites en fonctions des opérateurs standard.

2.2.4.1. Fonctions numériques et arithmétiques

Les fonctions numériques et arithmétiques que nous présentons prennent des entiers en argument ; chaque fonction renvoie un entier.

Les fonctions arithmétiques sont celles pour lesquelles un mot-clé ou un symbole est directement défini dans le langage. Ce sont les fonctions définies par le tableau 2.2. Les fonctions numériques ne sont accessibles qu'à travers un appel de fonction. La seule fonction numérique sur les entiers est la valeur absolue (`ABS`).

La définition de la plupart des fonctions arithmétiques est évidente ; seules, les définitions de la division et du modulo sont délicates et méritent d'être précisées.

```
IF (b = 0)
THEN
  Result := 0 ;
ELSE
  Result := a - (a/b)*b ;
END_IF ;
```

FIG. 2.3.: Définition en ST de `MOD(a,b)` selon [Lew98].

Division entière Selon la norme, « le résultat de la division de deux entiers doit être un entier de même type, avec troncation vers zéro »[5]. Il faut entendre par là que la division, en valeur absolue, est la division des deux nombres, tronquée à la virgule (donc `DIV(7, 3)` vaut 2).

Il reste à en définir le signe : la norme, ainsi que [Lew98], se bornent à illustrer par l'exemple le cas où le dividende est négatif : `DIV(-7, 3)` vaut `-2`. L'expérience sous ISaGRAF montre que cet interpréteur effectue un produit des signes des deux opérateurs : ainsi, `DIV(-4,3)` vaut `-1`, `DIV(4,-3)` vaut `-1`, `DIV(-4, -3)` vaut `1`.

Les différents manuels de référence ne contredisent pas cette définition. Il est à noter que, bien que cela ne soit pas spécifié par la norme, ISaGRAF provoque une erreur en cas de division par zéro.

Modulo En théorie, le modulo est le reste de la division entière. Cela signifie qu'il existe traditionnellement une relation

$$a = \texttt{DIV}(a, b) \cdot b + \texttt{MOD}(a, b)$$

Cette relation n'est pas contredite par la norme, qui définit le modulo d'après la division entière au moyen d'un programme ST. Ce programme est donné à la figure 2.3 et est équivalent à `Result := MOD(a,b)`.

Par conséquent, `MOD(12,3)` vaut `0`, `MOD(14,3)` vaut `2`, `MOD(-4,3)` vaut `-1`. Par continuité, on devrait avoir `MOD(4, -3)` égal à `1`, mais le cas d'un diviseur négatif n'est précisé par aucun des ouvrages de référence. Dans ce cas de figure (calcul de $\texttt{MOD}(x,y)$ où $y < 0$), l'interpréteur d'ISaGRAF donne invariablement le résultat `-1`. Le fait que le cas ne soit pas spécifié doit inciter à la prudence lors de l'utilisation de la fonction modulo avec des nombres négatifs.

2.2.4.2. Fonctions booléennes

Les fonctions booléennes sont `AND`, `OR`, `XOR`, `NOT`. Le symbole `&` peut être substitué à l'opérateur `AND`.

[5] D'après la version française de la norme [CEI93], tableau 24.

Fonction	Symbole
`GT`	`>`
`GE`	`>=`
`EQ`	`=`
`LE`	`<=`
`LT`	`<`
`NE`	`<>`

Tab. 2.3.: Fonctions de comparaison.

2.2.4.3. Fonctions de comparaison

Les fonctions de comparaison prennent deux arguments de même type (entier ou booléen) et renvoient un booléen. Le tableau 2.3 présente les fonctions de comparaison et leurs symboles associés.

2.3. IL

2.3.1. Présentation

Le langage IL est inspiré des langages de type assembleur. Un programme IL se lit ligne par ligne, chaque ligne étant composée d'une étiquette optionnelle, d'un opérateur et d'un argument optionnel.

Pour certaines commandes (notamment les commandes nécessitant deux arguments), le langage IL utilise un argument implicite. Cet argument, que nous appellerons *accumulateur*[6], est une variable interne à chaque fonction d'un programme IL, implicitement déclarée. Les opérations suivantes peuvent être effectuées sur l'accumulateur :

- chargement de la valeur d'une variable dans l'accumulateur ;
- enregistrement de la valeur de l'accumulateur dans une variable ;
- calcul d'une opération entre l'accumulateur et une variable.

Toute instruction de calcul renvoie le résultat du calcul dans l'accumulateur. Le langage IL propose de plus des instructions de saut : pour repérer la destination d'un saut, on définit une étiquette en début de ligne. Par exemple, la ligne de programme suivante déclenche une boucle infinie :

```
saut : LD  a
       JMP saut
```

2.3.1.1. Opérateurs du langage

Dans la suite, l'accumulateur sera noté a. L'accumulateur peut être booléen ou entier et n'est pas explicitement déclaré. Cependant, chaque opérateur induit une contrainte sur

[6] La norme désigne l'accumulateur sous le nom de *résultat courant*.

Instruction	Interprétation
LD s	$a \leftarrow s$
ST s	$s \leftarrow a$
LDN s	$a \leftarrow \neg s$
STN s	$s \leftarrow \neg a$
S s	$s \leftarrow$ TRUE si $a =$ TRUE
R s	$s \leftarrow$ FALSE si $a =$ TRUE

TAB. 2.4.: Opérateurs d'affectation (le symbole a représente l'accumulateur).

le type de l'accumulateur et celui de l'argument. Nous présentons dans cette section les opérateurs de notre fragment IL. Pour chaque opérateur, nous préciserons explicitement ses contraintes de typage.

Opérateurs d'affectation Les opérateurs d'affectation permettent de charger la valeur de l'accumulateur avec la valeur de l'argument, ou l'opération inverse. L'argument de l'opérateur `ST` doit être du même type que l'accumulateur. L'argument de l'opérateur `LDN` doit être booléen. L'argument des opérateurs `STN`, `S` et `R`, ainsi que l'accumulateur, doivent être de type booléen. Les opérateurs `LD` et `LDN` modifient le type de l'accumulateur, qui prend alors le type de l'argument. Les opérateurs d'affectation, et leur interprétation, sont donnés par le tableau 2.4.

Opérateurs de comparaison Les opérateurs de comparaison exigent que le type de l'argument soit le même que le type de l'accumulateur. Après l'exécution de l'instruction, l'accumulateur est de type booléen. Les opérateurs de comparaison sont les opérateurs de comparaison standard (voir le tableau 2.3). Par exemple `GT` s s'interprète par $a \leftarrow$ `TRUE` si $a > s$, $a \leftarrow$ `FALSE` sinon.

Opérateurs sur les booléens Les opérateurs sur les booléens requièrent que l'accumulateur et l'argument soient de type booléen. Les opérateurs sur les booléens sont les opérateurs booléens standard auxquels peut être ajouté le modificateur `N` (négation de l'argument). Ils sont donnés par le tableau 2.5 page suivante.

Opérateurs sur les entiers Les opérateurs sur les entiers bornés requièrent que l'accumulateur et l'argument soient de type entier borné. Les opérateurs sur les entiers sont les opérateurs arithmétiques standard (voir le tableau 2.2 page 26). Par exemple `ADD` s s'interprète par $a \leftarrow a + s$.

Opérateurs de saut Les instructions IL sont exécutées séquentiellement et à la suite, dans l'ordre du programme, sauf en cas d'instruction de saut. Dans le cas d'une instruction de saut, l'exécution reprend séquentiellement à partir de la destination du saut (l'étiquette

Instruction	Interprétation
`AND` s	$a \leftarrow a \wedge s$
`&` s	$a \leftarrow a \wedge s$
`ANDN` s	$a \leftarrow a \wedge \neg s$
`&N` s	$a \leftarrow a \wedge \neg s$
`OR` s	$a \leftarrow a \vee s$
`ORN` s	$a \leftarrow a \vee \neg s$
`XOR` s	$a \leftarrow (\neg a \wedge s) \vee (a \wedge \neg s)$
`XORN` s	$a \leftarrow (\neg a \wedge \neg s) \vee (a \wedge s)$

TAB. 2.5.: Opérateurs sur les booléens.

Instruction	Interprétation
`JMP` l	saut inconditionnel vers l'étiquette l
`JMPC` l	saut vers l'étiquette l si $a =$ `TRUE`
`JMPCN` l	saut vers l'étiquette l si $a =$ `FALSE`
`JMPNC` l	équivalent à `JMPCN` l
`RET`	arrêt de l'exécution pour le cycle en cours
`RETC`	arrêt de l'exécution si $a =$ `TRUE`
`RETCN`	arrêt de l'exécution si $a =$ `FALSE`
`RETNC`	équivalent à `RETCN`

TAB. 2.6.: Opérateurs de saut.

donnée en argument). Si l'opérateur contient le suffixe `C`, `CN` ou `NC`, l'accumulateur doit être booléen. Les instructions de saut sont données par le tableau 2.6.

2.3.1.2. Expressions parenthésées

Il est possible, en IL, de définir des « expressions parenthésées ». Par exemple, le programme suivant :

```
MUL( x
SUB  y
)
```

effectue l'opération $a \leftarrow a \cdot (\mathtt{x} - \mathtt{y})$. L'exécution de l'opérateur `MUL` est différée jusqu'à la rencontre de la parenthèse fermante. En fait, tout se passe comme si la valeur de l'accumulateur était placée dans une zone temporaire, et qu'un autre accumulateur était utilisé pour les calculs à l'intérieur de la parenthèse. le programme qui précède est d'ailleurs équivalent au suivant :

```
MUL(
LD   x
SUB  y
```

Programme	Interprétation
`MUL(`	$temp \leftarrow a$
`LD   x`	$a \leftarrow \mathtt{x}$
`SUB  y`	$a \leftarrow a - \mathtt{y}$
`)`	$a \leftarrow temp \cdot a$

TAB. 2.7.: Interprétation d'un programme à expression parenthésée.

```
)
```

Le tableau 2.7 détaille l'interprétation du programme.

Il est possible de combiner plusieurs expressions parenthésées en les empilant. Une parenthèse ouvrante peut être placée après tout opérateur sur les booléens et tout opérateur sur les entiers.

2.3.2. Syntaxe concrète

La syntaxe concrète du langage IL est présentée en annexe, à la section A.4 page 174.

2.3.3. Programmes biens formés

Toute déclaration suivie d'une liste de lignes de commande ne donne pas nécessairement lieu à un programme bien formé. Il existe quelques règles simples définissant quels programmes sont bien formés, et un compilateur IL exige que ces règles soient suivies. Ces règles sont les suivantes :

- on n'a pas déclaré deux variables de même nom ;
- toutes les variables sont déclarées ;
- on ne trouve pas en début de ligne deux étiquettes de même nom ;
- toutes les étiquettes figurant en argument d'un opérateur de saut sont déclarées (figurent en début de ligne) ;
- les arguments correspondent aux opérateurs (étiquettes pour les sauts, variable ou constante pour les calculs, etc.) ;
- toutes les contraintes de typage sont respectées ;
- si plusieurs expressions parenthésées sont utilisées, le parenthésage doit être cohérent :
  - à chaque parenthèse fermante doit correspondre une parenthèse ouvrante, et réciproquement,
  - les expressions parenthésées ne se chevauchent pas – elles s'empilent.

2.3.4. Langage IL_0

2.3.4.1. Définition

Le langage IL_0 est un fragment du langage IL, tel que défini à la section 2.3. Le fragment IL_0 a, par rapport à IL, les caractéristiques suivantes :

– tous les programmes de IL_0 sont bien formés ;
– il ne prend en compte que des valeurs booléennes ;
– il n'autorise pas l'usage des expressions parenthésées[7].

Ainsi, IL_0 ne prend pas en compte les types de données représentant des temporisations. Nous donnons dans cette section une syntaxe abstraite du langage IL_0, c'est-à-dire la définition formelle d'un programme écrit dans ce langage. Pour cela, nous décrivons P_{IL_0}, l'ensemble des programmes valides et bien formés du langage.

2.3.4.2. Syntaxe abstraite

Dans la suite, on désignera par S un ensemble de symboles correspondant aux identificateurs valides lexicalement (voir la section 2.2.1.2 page 23), et par $\mathbb{B}$ l'ensemble des constantes booléennes ($\mathbb{B} \stackrel{def}{=} \{\mathtt{TRUE}, \mathtt{FALSE}\}$).

Un programme $p \in P_{\mathrm{IL}_0}$ est un couple (Σ_p, C_p), où Σ_p est l'ensemble des variables, et C_p est le corps du programme p.

Variables L'ensemble Σ_p des variables se décompose en quatre sous-ensembles : les variables d'entrée, de sortie, d'entrée-sortie et internes, représentés respectivement par Σ_{in}, Σ_{out}, Σ_{io}, Σ_{int}.

Une variable $x \in \Sigma_p$ est un couple (s, c^i) où $s \in S$ est le nom de la variable et $c^i \in \mathbb{B}$ est sa valeur initiale.

On définit init_p la fonction qui, à un symbole de variable, associe sa valeur initiale (pour tout $(s, c^i) \in \Sigma_p$, $\mathrm{init}_p(s) = c^i$).

Corps du programme Le corps d'un programme IL_0 est formé d'une liste de commandes : $C_p = \{p_1, p_2, \ldots, p_k\}$. Le nombre k de commandes est appelé la taille du programme p. Chacune de ces commandes p_i est un couple (l_i, ins_i), où $l_i \in S \cup \{\varepsilon\}$ est une étiquette facultative[8], et où ins_i est une instruction.

On considère qu'une étiquette l est déclarée quand il existe un couple (l, ins_i) dans C_p. On définit L_p l'ensemble des étiquettes déclarées du programme p.

Instructions Une instruction ins_i est un couple (op_i, arg_i), où op_i est un opérateur standard d'affectation, de saut ou de calcul booléen (voir la description de ces opérateurs section 2.3.1.1 page 28), et où $arg_i \in \Sigma_p \cup L_p \cup \mathbb{B} \cup \{\varepsilon\}$. Pour simplifier l'écriture, une ligne d'instruction sera notée $op_i\ arg_i$ (par exemple `ST` s ou `JMPC` l).

[7]Il est cependant possible de simuler les expressions parenthésées en utilisant des variables temporaires pour l'accumulateur. Cette tâche peut être effectuée par le compilateur.
[8]Si aucune étiquette n'est définie pour la ligne i, alors $l_i = \varepsilon$.

Opérateur	Symbole
parenthésage	(*expression*)
appel de fonction	*nom* (*liste d'arguments*)
négation complément	`-` `NOT`
élévation à une puissance	`**`

multiplication
division
modulo
	`*`
	`/`
	`MOD`

addition soustraction	`+` `-`
comparaison	`<`, `>`, `<=`, `>=`
égalité inégalité	`=` `<>`
et logique	`AND`, `&`
ou exclusif	`XOR`
ou logique	`OR`

TAB. 2.8.: Opérateurs ST classés par ordre de priorité décroissante.

2.4. ST

2.4.1. Présentation

Il existe deux principales formes du langage ST : les expressions et les énoncés. Un programme ST est une séquence d'*énoncés* (*statements*) séparés par des points-virgules (`;`).

2.4.1.1. Expressions

Une expression est formée d'un des éléments suivants :
- une constante (booléenne ou entière) ;
- un opérateur unaire (`-`, `NOT`) appliqué à une expression ;
- un opérateur binaire appliqué à deux expressions ;
- une expression entre parenthèses ;
- un appel de fonction.

Par exemple, `((a+b) MOD c) / (ABS (d))` est une expression valide.

Le tableau 2.8 présente les opérateurs ST classés par ordre de priorité décroissante. Une expression ST peut être évaluée ; le résultat est une valeur, utilisable par exemple lors d'une affectation (voir la description de l'énoncé d'affectation ci-dessous).

2.4.1.2. Énoncés

Les énoncés ST sont les suivants :
- l'énoncé vide ;
- `EXIT` ;
- `RETURN` ;
- affectation ;
- sélection (`IF`, `CASE`) ;
- itération (`WHILE`, `REPEAT`, `FOR`).

Selon la définition de la norme le point-virgule est un terminateur (par opposition à séparateur). Cela signifie que tout énoncé doit toujours être suivi d'un point-virgule, y compris par exemple les énoncés de type `IF` ... `END_IF`.

Affectation L'énoncé d'affectation permet d'affecter à une variable la valeur d'une expression. Exemple :

```
a := 4+5 ;
```

Sélection L'énoncé IF permet d'effectuer une sélection entre deux énoncés, d'après la valeur d'une expression booléenne, par exemple :

```
IF a < b
THEN
  x := a ;
ELSE
  x := b ;
END_IF ;
```

La clause `ELSE` est optionnelle. Il existe également une clause `ELSIF` permettant de compacifier les programmes, comme l'illustre le tableau 2.9 page ci-contre. Un nombre arbitraire de clauses `ELSIF` peuvent être ajoutées à un énoncé `IF`.

L'énoncé `CASE` permet une sélection en fonction de la valeur d'une expression entière. Par exemple :

```
CASE a + b MOD 6 OF
  0,1 : result := a ;
  2 : result := b ;
  3..5 : result := a+b ;
END_CASE ;
```

Itération L'énoncé FOR permet d'effectuer un traitement en boucle en fonction d'un indice :

Avec `ELSIF`	Sans `ELSIF`

`IF a = b`
`THEN`
`  result := a+b ;`
`ELSIF a < b`
`THEN`
`  result := a ;`
`ELSE`
`  result := b ;`
`END_IF ;`
	`IF a = b`
	`THEN`
	`  result := a+b ;`
	`ELSE`
	`  IF a < b`
	`  THEN`
	`    result := a ;`
	`  ELSE`
	`    result := b ;`
	`  END_IF ;`
	`END_IF ;`

TAB. 2.9.: Exemple d'utilisation de la clause `ELSIF` : deux programmes équivalents.

```
FOR i := 1 TO 5 BY 2 DO
  a := a + 1 ;
END_FOR ;
```

Le mot-clé `BY`, optionnel, permet de définir l'incrément ; il peut être négatif. S'il n'est pas précisé, la valeur par défaut est `1`. Le contrôle de fin de boucle est effectué avant l'exécution du corps ; cela entraîne que le corps peut ne pas être exécuté du tout. Si, après incrémentation, la valeur de l'indice de boucle dépasse la valeur limite, l'exécution de l'énoncé `FOR` s'arrête. Dans ce cas, selon la norme, la valeur de l'indice de boucle est alors dépendante de l'implémentation. Il est en conséquence recommandé aux programmeurs de s'abstenir d'utiliser un indice de boucle en dehors de la boucle où il est défini.

L'énoncé `WHILE` effectue une itération en fonction d'une expression booléenne :

```
WHILE a < b DO
  a := a + 1 ;
END_WHILE ;
```

L'énoncé `REPEAT` effectue le même traitement que l'énoncé `WHILE`, mais évalue la condition de terminaison après avoir exécuté le corps :

```
REPEAT
  a := a + 1 ;
UNTIL a >= b
END_REPEAT ;
```

Énoncé `EXIT` L'énoncé `EXIT` est utilisé à l'intérieur d'un énoncé d'itération. Lorsqu'il est exécuté, il provoque l'arrêt de l'exécution de l'énoncé `WHILE`, `FOR` ou `REPEAT` le plus profond, comme l'illustre la figure 2.4 page suivante.

```
a := 0 ; b := 0 ;
WHILE a > 10 DO
  WHILE TRUE DO
    IF b >= 10
    THEN EXIT ;
    ELSE b := b + 1 ;
    END_IF ;
  END_WHILE ;
  a := a + 1 ;
END_WHILE ;
```

FIG. 2.4.: Exemple d'exécution de l'énoncé `EXIT`.

```
a := 0 ; b := 0 ;
WHILE a > 10 DO
  WHILE TRUE DO
    IF b >= 10
    THEN RETURN ;
    ELSE b := b + 1 ;
    END_IF ;
  END_WHILE ;
  a := a + 1 ;
END_WHILE ;
```

FIG. 2.5.: Exemple d'exécution de l'énoncé `RETURN`.

Énoncé `RETURN` L'énoncé `RETURN` provoque l'arrêt de l'exécution du bloc fonctionnel, comme l'illustre la figure 2.5.

2.4.2. Syntaxe concrète

La syntaxe des programmes ST est présentée à la section A.5 page 175.

2.4.3. Programmes biens formés

Nous définissons les règles de bonne formation suivantes pour un programme ST :

- un programme ST comporte au moins un énoncé. Il peut cependant s'agir d'un énoncé vide (formulé syntaxiquement au moyen d'un point-virgule seul) ;
- toute variable utilisée dans un programme doit avoir été déclarée.

Nous y ajoutons les règles suivantes, permettant de proscrire des cas non spécifiés par la norme, ou déclarés comme dépendant de l'implémentation :

- les indices des boucles `FOR` ne sont pas utilisés en dehors de leur boucle ;

– tout énoncé `EXIT` doit être situé dans un sous-terme d'un énoncé `WHILE`, `FOR` ou `REPEAT`.

2.4.4. Langage ST_0

2.4.4.1. Définition

De même que le langage IL_0, le langage ST_0 est un fragment du langage ST, tel que défini plus haut. Le fragment ST_0 a pour caractéristique principale de ne prendre en compte que les valeurs booléennes. Par conséquent , il ne prend pas en compte les énoncés `FOR` et `CASE`, non applicables dans un contexte uniquement booléen. Par ailleurs, l ne prend pas non plus en compte l'énoncé `REPEAT`, qui peut être trivialement simulé à l'aide d'un énoncé `WHILE`. On appellera P_{ST_0} l'ensemble des programmes valides du langage ST_0.

2.4.4.2. Syntaxe abstraite

Un programme $p \in P_{\mathrm{ST}_0}$ est un terme. Nous définissons inductivement chacune des deux formes de base du langage ST_0 : les expressions et les énoncés.

Expressions Chacun des termes suivants est une expression e valide :
– les constantes booléennes `TRUE` et `FALSE` ;
– une variable x ;
– la négation d'une expression $\neg e'$;
– la conjonction de deux expressions $e' \wedge e''$;
– la disjonction de deux expressions $e' \vee e''$.

Énoncés Chacun des termes suivants est un énoncé s valide :
– **return** ;
– **exit** ;
– l'énoncé vide **skip** ;
– l'affectation d'une expression à une variable $x \leftarrow e$;
– la sélection de deux énoncés par une expression **if** e **then** s' **else** s'' ;
– l'itération d'un énoncé par une expression **while** e **do** s' ;
– la séquence de deux énoncés $s' \triangleright s''$.

3. Modèles des langages de la norme

3.1. Introduction

Nous présentons ci-après un inventaire de travaux visant à la conception sûre de programme d'API, en rapport avec les langages de la norme IEC 61131-3. On constate qu'il existe deux démarches complémentaires :

- une démarche consiste à définir des modèles théoriques adaptés à la conception de systèmes sûrs, c'est-à-dire pour lesquels des méthodes de validation existent. Dans le cas des langages de la norme IEC 61131-3, des méthodes ont été developpées pour permettre l'extraction automatique de programmes conformes à partir de tels modèles. Cette approche, qui met l'accent sur la méthodologie, permet d'apporter, à long terme, des méthodes de conception sûre, mais nécessite de la part de l'industrie une évolution, parfois majeure, de ses pratiques quotidiennes. Nous qualifierons les modèles ainsi défini de modèles dédiés à la « conception de programmes sûrs » ;
- une autre démarche consiste à s'adapter aux langages de la norme, utilisés massivement dans l'industrie, et à développer des techniques de validation appropriées. Cette approche, à plus court terme, permet, quels que soient l'origine, le style et la méthode de conception du programme, d'aider à y trouver des *bugs*. Le travail présenté dans cette thèse, motivé par une demande industrielle, relève de cette seconde approche, qui consiste à définir une sémantique d'un langage de programmation dans un formalisme donné, puis de définir ensuite des méthodes de validation sur ce formalisme. Il est alors possible de concevoir un outil permettant la vérification automatique du programme de départ. Celui-ci ne fait l'objet d'aucune (ou presque) restriction particulière. Les modèles ainsi obtenus seront désignés sous le nom de « modèles pour la vérification de programmes existants ».

La première approche est qualifiée de « long terme », car elle n'est pas toujours directement applicable, pour les raisons suivantes, évoquées pour la plupart par [FL00] :

- la plupart des programmeurs d'API n'ont pas de compétences en méthodes formelles, et utilisent leurs techniques de programmation traditionnelles ;
- en cas de volonté de faire évoluer les méthodes de conception, le coût de mise en place d'une telle solution (formation et montée en compétence des ingénieurs, embauche d'experts, travail de modélisation des produits existants...) peut être prohibitif ;
- il existe des millions de programmes d'API qui ne peuvent être traités formellement que par la vérification.

Nous présentons dans ce chapitre quelques exemples de modèles existants, relevant de chacune des deux approches. Nous nous inspirons, en les complétant et en nous focalisant sur les langages qui font l'objet de notre étude, d'autres inventaires existant dans la littérature [LCRRL99, FL00, Mad00, MW00].

3.2. Modèles existants

Nous présentons dans un premier temps des modèles ayant été définis de manière complète dans trois formalismes particuliers : les réseaux de Petri, les langages synchrones et les PLC-automates. Nous évoquerons ensuite d'autres travaux, en les distinguant en fonction de l'approche sur laquelle ils sont fondés.

3.2.1. Modèles à base de réseaux de Petri

Les réseaux de Petri sont parfois utilisés pour effectuer la spécification de systèmes automatisés. Afin de les adapter au domaine particulier des API, une extension des réseaux de Petri, les réseaux de Petri interprétés, a été proposée par Frey et Litz [FL98]. Dans ce modèle sont directement intégrées la gestion du cycle et des entrées-sorties. Une extension aux réels a été définie dans [Fre00b], qui propose une traduction vers SFC. Une traduction vers IL (et LD) a été proposée dans [Fre00a]. Ces travaux, qui s'inscrivent dans l'approche de la génération de programmes sûrs, permettent de développer des outils de conception de systèmes pour API, en exploitant les outils existants pour la validation de systèmes décrits par des réseaux de Petri.

Symétriquement aux précédents, d'autres travaux ont visé à définir une sémantique à base de réseaux de Petri aux programmes écrits dans les langages de la norme, dans le but de modéliser des programmes existants. Une telle sémantique, pour le langage IL, est par exemple proposée par Heiner *et al* [HM98b, HM98a]. Les auteurs définissent pour cela un fragment du langage IL, qu'ils nomment IL_0, et qui possède les caractéristiques suivantes :
- il comprend les commandes de base IL ;
- il prend en compte le type booléen, et les entiers et structures de données, d'une taille limitée à 8 bits ;
- il ne prend pas en compte les temporisations.

Par rapport à celui que nous avons défini à la section 2.3.4, le fragment étudié par Heiner *et al* propose la prise en compte d'entiers et de structures de données, mais impose une limitation sur leur taille. Cela est dû au fait que des réseaux de Petri ont été définis pour chaque opération sur 8 bits (de manière non paramétrable). Les auteurs définissent plusieurs sémantiques (dont une sémantique opérationnelle ; celle présentée à la section 4.2.3 page 49 repose sur les mêmes principes) mais n'établissent pas de lien explicite entre elles. Une sémantique du langage en termes de réseaux de Petri est définie. Un réseau est proposé pour chaque commande IL, puis la composition des réseaux entre eux est définie. Le modèle (appelé « programme utilisateur ») est complété par un modèle représentant l'environnement (gestion des entrées-sorties, appelé « programme système »), qui est présenté au travers d'un exemple. Les auteurs présentent la composition entre les deux modèles système et utilisateur, qui permet de définir le cycle. Aucune méthode de vérification n'est présentée sur le modèle.

À partir des travaux précédent, Mertke et Menzel, dans [MM00], décrivent la mise en œuvre d'un environnement de programmation/validation, conçu dans un but de commodité d'emploi. Les recherches des auteurs portent en particulier sur un langage simple permettant de décrire des propriétés, sans manipuler de logique temporelle. Un tel langage se compose de structures de phrases élémentaires, comme la construction « si...alors... »[1], associées à des mots-clés exprimant 3 modalités (exigence, interdiction, possibilité)[2].

3.2.2. Utilisation des langages synchrones

Jiménez-Fraustro et Rutten [JFR99a, JFR99b, JF01] proposent une traduction de ST dans les langages synchrones, en particulier Signal. Le but de ces travaux est d'intégrer la prise en compte de programmes ST existants dans des environnements de validation. Les langages synchrones considèrent les données (les variables) comme des signaux ; chaque signal est une suite de valeurs, cadencée par une horloge qui lui est propre. L'environnement Signal offre en particulier l'opération de sur-échantillonnage, qui permet de sélectionner certaines valeurs d'un signal selon une cadence donnée. Cette fonctionnalité permet, entre autres, de modéliser simplement la faculté d'observer des variables à deux niveaux : interne (tout au long de l'exécution du programme) ou externe (uniquement lors de la phase de transmission des sorties). Cette approche permet de simplifier l'écriture des propriétés sur le comportement du système.

Les travaux des auteurs, en particulier [JF01], décrivent la traduction de programmes ST vers Signal. La traduction en elle-même, bien que complexe, n'est pas prouvée (elle est néanmoins testée sur des exemples). Le but de cette approche est d'intégrer la spécification ainsi obtenue à des outils de validation utilisant la technologie synchrone (comme par exemple l'outil de simulation Simulink [JFR00]).

3.2.3. Un modèle temporisé : les PLC-automates

Certains modèles proposent une prise en compte directe du temps. Cette prise en compte est motivée par le fait que certaines applications sont fortement dépendantes de délais imposés à leurs actions. Le respect (en général le non-dépassement) de ces délais doit être validé. D'autre part, une problématique typique des API est celle du temps de cycle : de par la présence de nombreuses instructions, de boucles ou de délais, le temps d'exécution d'un cycle d'API est variable. Cependant, ce temps doit être borné : par exemple, si une entrée est modifiée deux fois (par exemple pour changer furtivement de valeur, et reprendre sa valeur précédente) avant que la phase d'acquisition ait lieu, la valeur transitoire est manquée par l'API. Pour ce problème, il n'existe pas de meilleure solution que de s'assurer

[1] Selon le papier, « d'autres structures ont été définies, afin de donner à l'utilisateur un ensemble de types de phrases redondants, de telle sorte qu'il puisse choisir son style de phrases favori ».

[2] En anglais *demands*, *prohibitions*, *possibilities*.

que le cycle de l'API soit suffisamment rapide[3]. Cela nécessite d'estimer correctement le temps d'exécution du programme, puis de décider quel modèle d'API doit être employé (quelle puissance nécessaire, donc quel budget) pour équiper le système. C'est entre autres pour résoudre ce problème que le modèle des PLC-automates a été conçu.

Les PLC-automates, proposés par Dierks dans [Die97a], sont un formalisme dédié à la conception de systèmes pour API, par la génération automatique de programmes sûrs. Il s'agit d'un modèle à base de machines à états, temporisé, pour laquelle une représentation graphique est définie. Cette représentation graphique est destinée à être utilisée comme langage de spécification, à l'aide d'un environnement dédié. L'auteur définit une sémantique des PLC-automates à l'aide de la logique DC[4]. Il est possible de spécifier un système à l'aide de formules écrites dans cette logique, d'après la spécification informelle du système et son cahier des charges. La spécification sous forme de PLC-automates peut être déduite semi-automatiquement de ces formules (la traduction est décrite dans [Die97b]). Les auteurs évoquent une traduction automatique des PLC-automates vers du code ST, qui permet ainsi d'extraire automatiquement un programme d'API à partir de la spécification ainsi obtenue.

Une sémantique des PLC-automates en termes d'automates temporisés est définie dans [DFMV98]. Le papier fournit une preuve de la correspondance avec la sémantique DC. Ce résultat permet, à partir de la spécification du système donnée sous la forme de PLC-automates, d'effectuer son codage vers le format d'entrée d'un model-checker temporisé.

Les résultats de ces recherches ont fait l'objet d'une implémentation, sous la forme du logiciel MOBY/PLC [DT00]. Ce logiciel est un environnement de conception de programmes d'API, qui propose :

- un éditeur graphique, intégrant la représentation des PLC-automates, permettant de définir le système selon les méthodes évoquées ci-dessus ;
- un simulateur ;
- un traducteur vers les formats des model-checkers Kronos [Yov97] et Uppaal [LPY97] ;
- un générateur de code ST.

Des extensions ont ensuite été proposées, comme par exemple la prise en compte de diagrammes à contrainte par Olderog dans [Old99], qui en définit pour cela une sémantique DC adaptée.

Les travaux autour des PLC-automates définissent un large cadre pour leur utilisation ; cependant, la prise en compte de programmes existants n'est pas offerte.

3.2.4. Modèles pour la génération de programmes sûrs

Il existe de nombreux autres travaux évoquant la génération de programmes d'API à partir d'une spécification formelle. Nous citerons par exemple Egger *et al* [EFP94], qui proposent la définition d'un langage (un sous-ensemble de ST) pour lequel une sémantique

[3] Il est possible de donner un effet de rémanence aux entrées, mais cela pose d'autres problèmes : il faut alors refuser, ou retarder, certaines valeurs.

[4] Le *Duration Calculus*, introduit par Chaochen *et al* [CHR91], est une logique temporelle permettant d'exprimer des propriétés sur des intervalles de temps.

(dénotationnelle) est définie. Les auteurs proposent un compilateur qui permet d'assurer que la sémantique est bien représentée dans le langage-machine cible. La preuve en elle-même n'est pas fournie. Le but de ce travail est de lever les ambiguïtés et les incohérences de la norme, en définissant un langage « propre ».

Nous citerons également les travaux de Bates *et al*, qui définissent dans [BCK98a] une traduction de Statecharts vers du code IL. La traduction en elle-même est présentée, mais n'est pas étayée par des preuves ou des arguments formels. La méthode est illustrée dans [BCK98b] par une étude de cas.

Enfin, Kowalewski [KES99] propose l'utilisation des systèmes conditions/événements temporisés pour la spécification d'un système avec API. Les auteurs évoque une traduction du modèle dans des automates temporisés, et présente un outil, Verdict, permettant la conception/validation de systèmes pour API. L'outil permet de spécifier un système à l'aide d'un éditeur graphique, et propose l'intégration, pour certaines parties du programme, de portions de code IL, par l'intermédiaire d'un traducteur. Le papier présente l'outil Verdict comme un atelier de modélisation : son utilisateur définit le modèle de son système, dont le logiciel sait extraire une spécification dans le format d'entrée d'un model-checker temporisé (Kronos et Hytech). Il n'est pas fait mention d'exportation du modèle vers un langage-cible, ni d'importation à partir d'un programme complet existant. Tel que présenté, l'atelier ne permet pas de relier automatiquement le modèle avec du code programme.

3.2.5. Modèles pour la vérification de programmes existants

Mader et Wupper [MW99] ont défini une sémantique opérationnelle IL à base d'automates temporisés. Le fragment IL étudié est similaire à notre IL_0 (section 2.3.4 page 31) ; il y ajoute un type de données représentant des temporisations. Les calculs de délais sont représentés par l'intermédiaire du bloc fonctionnel spécialisé TON (défini par la norme). La vérification effective de propriétés n'est pas décrite, mais la sémantique définie est utilisable pour la validation de programmes à l'aide d'un model-checker temporisé (Kronos, Uppaal).

Notons que d'autres modèles ont été développés, dans le but d'utiliser d'autres méthodes de validation. Bornot *et al* [BHLL00a] proposent une analyse statique de programmes IL : la méthode consiste à caractériser les intervalles de données associés à chaque variable et à l'accumulateur. Les auteurs proposent également une caractérisation de l'accumulateur au moyen d'une expression (p. ex. $X + Y$, ou $X \leq 10$), dont les intervalles de chaque variable sont déduits. Cette approche permet une vérification de propriétés approximatives ou très simples (division par zéro...), dont la complexité est triviale. Les auteurs souhaitent utiliser ces travaux en combinaison avec une traduction de IL vers SMV.

Expériences de validation par le model-checking symbolique Plusieurs traductions des langages de la norme vers SMV ont été proposées. Ces traductions ne définissent en général pas de sémantique au préalable. Le papier fondateur en la matière est celui d'Il Moon [Moo94], qui propose une traduction des programmes LD en SMV. Il ne prend pas en compte les sauts, autorisés en LD. Indépendamment du précédent, Rausch et Krogh

[RK98] proposent une traduction similaire de systèmes décrits partiellement sous la forme de réseaux condition/événement, et partiellement en LD. Là encore, les sauts ne sont pas pris en compte. Des approches similaires, hors du cadre de ce document, ont été développées pour d'autres langages de la norme (en particulier SFC).

De manière plus complète, Bornot *et al* [BHLL00b] proposent un codage de SFC en SMV. Dans ce codage, chaque transition du système représente l'exécution d'un cycle complet. La hiérarchie entre plusieurs sous-parties du systèmes, elles-mêmes représentées en SFC, est respectée : les sous-parties sont représentées par des modules. Le catalogue de propriétés vérifiées est assez complet, et couvre de nombreux types de propriétés fréquemment utilisées (atteignabilité, vivacité, exclusion mutuelle, dépendance causale...).

3.3. Présentation de notre approche

Nos travaux visent à définir des méthodes complètes de validation de programmes d'API existants. Nous définissons pour cela un modèle complet des programmes, puis nous donnons un cadre pour l'utilisation de ce modèle, en traitant des exemples à l'aide du model-checker symbolique SMV. Nous ne représentons pas directement les temporisations. En particulier, les programmes qui ont fait l'objet de notre travail ne sont pas temporisés. Nous proposons cependant une approche (section 7.2.1.1 page 141) permettant de vérifier certaines propriétés paramétrées par des délais[5].

Modélisation Nous définissons une sémantique opérationnelle des langages étudiés (IL et ST). En effet, même s'il est possible, par exemple, de définir directement une traduction de ST en SMV, une telle traduction pose les problèmes suivants :
- sa validation est problématique si le langage comporte des structures complexes (typiquement : les sauts, ou les `EXIT` et `RETURN` de ST) ;
- elle est totalement dépendante du model-checker utilisé.

La définition d'une sémantique dans un formalisme adapté (chapitre 4), puis du codage de celle-ci dans un model-checker (chapitre 6), permet de s'affranchir de la dépendance avec un model-checker donné. Par ailleurs, la sémantique permet de simplifier la validation de la traduction, en la rendant plus claire.

Exploitation Une sémantique opérationnelle utilisant des systèmes de transitions a pour conséquence un nombre élevé d'états. Il existe cependant des techniques permettant de vérifier efficacement de tels systèmes. Nous avons choisi, pour notre étude de cas, le model-checking symbolique car, contrairement au model-checking classique, celui-ci permet d'évi-

[5] Notons de plus que, même si les langages IL et ST permettent de manipuler des types représentant des temporisations, il n'existe dans ces langages aucune méthode directe pour effectuer une attente : celle-ci est effectuée sous la forme d'un appel au bloc fonctionnel TON, défini par la norme. Dans le cadre du projet Vulcain, celui-ci a été modélisé par O. Rossi [RS00] de la manière décrite ci-dessus, dans le formalisme modulaire présenté au chapitre suivant.

ter la représentation explicite des états en mémoire. Par conséquent, la complexité de la vérification est bien moins dépendante du nombre d'états [JEK+92].

Expérimentation Nous avons défini un codage de la sémantique opérationnelle dans le format d'entrée du model-checker symbolique SMV (décrit au chapitre 6), puis appliqué ce codage sur des exemples (chapitre 7). Notre démarche reste cependant indépendante du model-checker utilisé, et peut ainsi s'adapter à l'évolution des techniques.

4. Sémantique opérationnelle des langages textuels

4.1. Introduction

4.1.1. Généralités

La sémantique opérationnelle d'un langage de programmation est une description formelle du comportement des programmes écrits dans ce langage, fréquemment représentée à l'aide de systèmes de transitions. La sémantique opérationnelle est donc une méthode permettant d'associer un système de transition à un programme bien formé.

Les systèmes de transitions (voir par exemple [Arn92]) sont largement utilisés dans le domaine de la spécification et la vérification automatique, à laquelle ils sont particulièrement adaptés, en permettant la description entièrement formelle, et non ambiguë, du comportement d'un système automatisé (un programme dans notre cas). Les formalismes à base de systèmes de transitions sont nombreux ; il y a donc un choix à faire. Les objectifs du choix d'un formalisme sont les suivants :

– la sémantique exprime précisément et complètement le comportement d'un programme ;
– la description du comportement est aussi simple que possible, pour en faciliter la compréhension et la maîtrise ;
– le modèle est commun entre plusieurs langages et prévoit leur interaction ;
– le modèle formel associé à un programme est performant du point de vue de son exploitation (par exemple pour en effectuer efficacement la vérification) ;
– le modèle est modulaire.

Le fait qu'un modèle formel soit performant signifie, entre autres, que l'on peut en effectuer un codage dans un langage de spécification, de manière aussi naturelle et traçable que possible, afin de garantir une exploitation efficace des résultats. Dans ce but, la compacité du modèle joue un rôle important, en permettant naturellement un codage efficace. Un modèle moins compact engendrera un codage de grande taille, dont le traitement sera plus difficile.

La prise en compte de plusieurs langages, combinés entre eux, était l'un des objectifs du projet Vulcain. Nous serons ainsi amenés à proposer un modèle formel adapté à la prise en charge de langages différents.

Nous proposerons dans ce chapitre deux modèles pour représenter la sémantique opérationnelle des langages textuels. Alors que le langage IL reste relativement simple, nous examinerons les difficultés posées par la prise en compte de ST.

4.1.2. Modéliser l'interprétation

Pour chacun des langages et chacun des formalismes choisis, nous présentons une sémantique comprenant :

- un modèle du comportement du programme ;
- l'intégration de ce programme dans le fonctionnement de l'API.

Cela consiste à définir un modèle global, définissant le comportement de l'ensemble (programme, API) dans un seul formalisme. Plus précisément, nous serons amenés à modéliser explicitement le cycle de l'API. Comme nous l'avons vu à la section 1.2.1.2 page 14, le cycle de l'API se compose de trois phases :

- acquisition des entrées ;
- exécution du programme ;
- affectation des sorties.

Puisque nous nous proposons de modéliser et d'étudier le comportement des programmes d'API, avec une granularité très fine, nous ne modéliserons pas les appareils recevant le signal en sortie. Par conséquent, la phase d'affectation des sorties ne sera pas modélisée en soi. Il reste bien entendu possible d'observer les variables de sortie, avant et après la supposée phase d'affectation. Pour cela, nous distinguerons une phase de *fin de cycle*, qui indique que le programme est arrivé en fin d'exécution. Cette phase nous servira de point d'observation.

Dans un système de transitions, les transitions représentent généralement des actions et les états des points d'observation. Par exemple, la phase de fin de cycle est un point d'observation ; l'acquisition des variables, la boucle de la fin d'un cycle vers le début d'un nouveau, ou l'exécution d'une instruction du programme, sont des actions.

En résumé, une simulation du modèle traverse les étapes suivantes :

1. (action) acquisition des entrées ;
2. (action) exécution des instructions du programme ;
3. (observation) fin de cycle ;
4. (action) boucle vers le cycle suivant.

On note que les étapes 1 et 4 peuvent être effectuées en parallèle. En fonction des modèles proposés, nous serons parfois amenés à les associer dans la même transition.

4.2. Sémantique « directe »

4.2.1. Introduction

Nous proposons dans un premier temps une sémantique opérationnelle à partir de systèmes de transitions non étiquetés. Ce format est proche de celui qu'utilisent bon nombre de model-checkers, parmi lesquels SMV (voir la section 5.4 page 108). La sémantique présentée dans cette section permet d'associer à tout programme IL_0 ou ST_0 son interprétation dans le formalisme utilisé. Cette interprétation est définie de manière globale, au vu du programme entier. Elle a par conséquent le défaut de ne pas être modulaire, mais a l'avantage de se coder presque directement dans le model-checker.

4.2.2. Définition du formalisme

Nous considérerons dans cette section des systèmes de transitions simples de la forme $A = \langle Q, \rightarrow, I \rangle$, où Q est un ensemble d'états (aussi appelés *configurations*) dont la forme dépend du langage de départ. La relation de transition $\rightarrow$ est un sous-ensemble de $Q \times Q$, et $I \subseteq Q$ est l'ensemble des états initiaux.

4.2.3. IL_0

On considère un programme IL_0 constitué de k lignes. On rappelle que Σ est l'ensemble des variables du programme, et que les variables sont des couples (s, c^i) composés d'un symbole et d'une valeur initiale (voir la section 2.3.4.2 page 32). On appelle *valuation* de Σ une fonction associant à chaque symbole de variable s la valeur courante (`TRUE` ou `FALSE`) de la variable qu'il représente.

L'ensemble Q des états du système de transitions est constitué de triplets de la forme (V, a, m), tels que :

- V est une valuation de Σ ($V \in \mathbb{B}^\Sigma$) ;
- $a \in \{\mathtt{FALSE}, \mathtt{TRUE}\}$ est la valeur de l'accumulateur ;
- $m \in \{1, \ldots, k+1\}$ est le compteur de programme.

On définit $I \subseteq Q$, l'ensemble des *états initiaux* : (V, a, m) est initial si et seulement si le compteur de programme m vaut $k+1$ et si, pour tout variable (s, c^i), $V(s) = c^i$.

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

4.2.3.1. Définition des transitions

Il reste à définir quelles transitions existent entre les états de Q (le composant $\rightarrow \subseteq Q \times Q$ de A). À cet effet, chaque ligne de programme donne lieu à un ensemble de transitions. 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 un non-déterminisme.

$op\,s$	$\mathrm{b}_{op}(a,s)$
`AND` s	$a \wedge V(s)$
`ANDN` s	$a \wedge \neg V(s)$
`&` s	$a \wedge V(s)$
`&N` s	$a \wedge \neg V(s)$
`OR` s	$a \vee V(s)$
`ORN` s	$a \vee \neg V(s)$
`XOR` s	$(\neg a \wedge V(s)) \vee (a \wedge \neg V(s))$
`XORN` s	$(\neg a \wedge \neg V(s)) \vee (a \wedge V(s))$

TAB. 4.1.: Valeur de l'accumulateur après une instruction sur les booléens, pour tout a et pour tout s.

Pour la ligne i, ces transitions s'appliquent aux états (V, a, i) où le pointeur de programme est justement sur la ligne i. En général, l'état courant sera modifié en fonction de la nature de l'instruction et on passera à la ligne $i+1$. Nous notons ins_i l'instruction de la ligne i du programme IL.

Opérateurs d'affectation Nous définissons, pour tout a et pour toute valuation V, les transitions en fonction de l'instruction ins_i :

`ST` s on crée une transition $(V, a, i) \to (V', a, i+1)$ avec $V'(s) = a$ et, pour toute variable (x, c^i) avec $x \neq s$, $V'(x) = V(x)$;

`LD` s on crée une transition $(V, a, i) \to (V, V(s), i+1)$;

`LDN` s on crée une transition $(V, a, i) \to (V, \neg V(s), i+1)$;

`STN` s on crée une transition $(V, a, i) \to (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = \neg a$;

`S` s on crée une transition $(V, a, i) \to (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = \texttt{TRUE}$ si $a = \texttt{TRUE}$ et $V'(s) = V(s)$ sinon ;

`R` s on crée une transition $(V, a, i) \to (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = \texttt{FALSE}$ si $a = \texttt{TRUE}$ et $V'(s) = V(s)$ sinon.

Opérateurs sur les booléens Dans le cas où $ins_i = op\,s$ et op est un opérateur sur les booléens, pour tout a et pour toute valuation V, on crée une transition $(V, a, i) \to (V, \mathrm{b}_{op}(a, s), i+1)$, où b_{op} est défini comme décrit par le tableau 4.1.

Opérateurs de saut On note i_l la ligne de commande correspondant à l'étiquette l. Nous définissons, pour tout a et pour toute valuation V, les transitions en fonction de l'instruction ins_i :

`JMP` l on crée une transition $(V, a, i) \to (V, a, i_l)$;

`JMPC` l — si $a =$ `TRUE`, on crée une transition $(V, a, i) \to (V, a, i_l)$,
si $a =$ `FALSE`, on crée une transition $(V, a, i) \to (V, a, a + 1)$;

`JMPCN` l — si $a =$ `TRUE`, on crée une transition $(V, a, i) \to (V, a, i_l)$,
si $a =$ `FALSE`, on crée une transition $(V, a, i) \to (V, a, a + 1)$;

`RET` — on crée une transition $(V, a, i) \to (V, a, k + 1)$;

`RETC` — si $a =$ `TRUE`, on crée une transition $(V, a, i) \to (V, a, k + 1)$,
si $a =$ `FALSE`, on crée une transition $(V, a, i) \to (V, a, a + 1)$;

`RETCN` — si $a =$ `TRUE`, on crée une transition $(V, a, i) \to (V, a, a + 1)$,
si $a =$ `FALSE`, on crée une transition $(V, a, i) \to (V, a, k + 1)$.

4.2.3.2. Lecture des entrées

Enfin on crée des transitions modélisant la phase de lecture des entrées, qui intervient juste après la fin du cycle de l'API. Nous considérons que les nouvelles valeurs des entrées peuvent prendre des valeurs arbitraires : pour toutes valuations V et V' ne différant que sur leurs variables d'entrées (c.-à-d. pour tout $(s, c^i) \in \Sigma \setminus \Sigma_{in}$, on a $V'(s) = V(s)$) on crée une transition de $(V, a, k + 1)$ vers $(V', a, 1)$. Ces transitions permettent également de simuler le nouveau cycle (voir la description du cycle de l'API à la section 1.2.1.1 page 13).

4.2.3.3. Exemple

Nous illustrons la sémantique opérationnelle sur un exemple inspiré de [Mel01]. Après sa description, nous indiquons un programme qui l'implémente, puis nous donnons la sémantique opérationnelle de ce programme.

Présentation Dans un moteur, on utilise un réservoir d'huile. Le niveau de l'huile que contient le réservoir diminue en permanence. Pour ré-alimenter le réservoir en huile, le moteur d'une pompe se déclenche lorsque le niveau est descendu en-dessous d'un seuil minimal. Quand le niveau d'huile est remonté à un niveau maximal, le moteur de la pompe s'arrête. L'intervalle de niveau d'huile est déterminé au moyen de deux jauges (figure 4.1 page suivante).

On utilise un automate industriel pour commander le moteur. Les deux jauges envoient chacune une valeur en entrée dans l'automate. Il s'agit d'une valeur booléenne égale à `TRUE` si la jauge est à sec, et à `FALSE` si la jauge trempe. L'automate envoie une valeur en sortie, à `TRUE` pour mettre le moteur en marche, à `FALSE` pour l'arrêter.

On ne se souciera pas, dans cet exemple, des données reçues en entrée qui correspondraient à des anomalies physiques (par exemple, le cas où la jauge du bas est à sec, mais pas celle du haut). De manière générale, un bloc fonctionnel est appelé par un programme principal qui a pour charge de contrôler ce type de problèmes (et qui peut lui-même être vérifié à son tour).

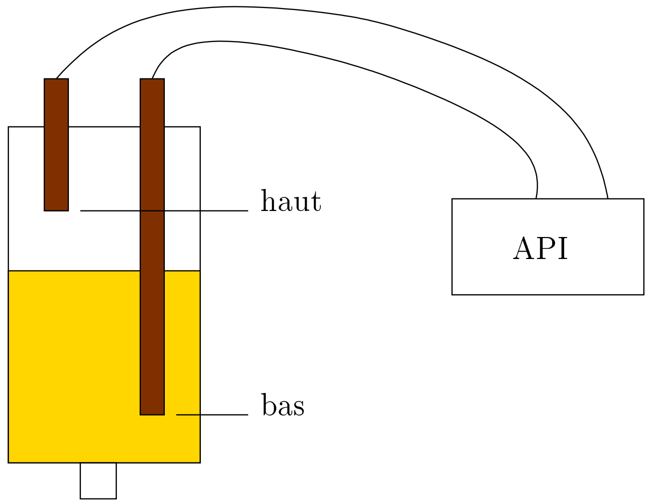

Fig. 4.1.: Réservoir d'huile

Variables Le système, entièrement booléen, utilise les variables suivantes :

`bas_sec` capteur du bas, il vaut `TRUE` s'il est à sec (entrée) ;

`haut_sec` capteur du haut, même fonctionnement (entrée) ;

`moteur` actionneur du moteur, une valeur de `TRUE` déclenche le moteur (sortie).

Programme Pour réaliser cette fonction, un opérateur a écrit le programme IL de la figure 4.2 page ci-contre. Une variable supplémentaire a été ajoutée, afin de pouvoir lire l'ancienne valeur de la variable `moteur` d'un cycle sur l'autre (les variables de sortie ne sont pas accessibles en lecture, voir la section 2.2.3 page 25) ; la variable `moteur_mem` sert donc de tampon).

Sémantique opérationnelle du programme Dans notre exemple, la taille k du programme est égale à 5. Le système de transitions associé à l'exemple du réservoir d'huile contient 56 états accessibles[1]. Il n'est pas utile de le représenter dans son intégralité, mais nous en décrivons quelques parties significatives.

Acquisition des entrées À partir de chaque état q « début de cycle », toutes les transitions possibles ayant q pour origine ont pour extrémité un état ne différant de q que par les variables d'entrée, et par le paramètre de ligne (égal à 0 dans q et à 1 dans les états extrémités des transitions). Par exemple, si q est l'état $(V, \texttt{FALSE}, 0)$ où $V(s) = \texttt{FALSE}$ pour tous les symboles s des variables déclarées, l'ensemble des transitions possibles à partir de

[1] Les variables `moteur` et `moteur_mem` contenant, en début de cycle, toujours la même valeur, le problème dépend de trois paramètres booléens (`moteur_mem`, `haut_sec`, `bas_sec`). Chaque cycle contient 7 étapes (acquisitions des entrées, boucle, exécution de 5 lignes de code). Le nombre d'états accessibles est donc $2^3 \cdot 7 = 56$.

```
VAR_INPUT
        bas_sec : bool ;
        haut_sec : bool ;
END_VAR

VAR_OUTPUT
        moteur : bool := FALSE ;
END_VAR

VAR
        moteur_mem : bool := FALSE ;
END_VAR

LD      moteur_mem
OR      bas_sec
AND     haut_sec
ST      moteur_mem

ST      moteur
```

FIG. 4.2.: Programme IL implantant l'exemple du réservoir d'huile.

q est présenté figure 4.3 page suivante. On rappelle que les transitions d'acquisition sont les seules transitions non-déterministes de notre système.

Détail de l'exécution du programme La figure 4.4 page 55 met en correspondance l'évolution de deux exécutions extraites du système de transitions, et les lignes de commande du programme du réservoir, à partir de deux situations différentes : une menant à déclencher le moteur (les deux jauges sont à sec), et une où le moteur est en route, mais où il n'est pas encore temps de l'arrêter (la jauge haute est la seule à sec, et le moteur tourne déjà). Pour simplifier le graphique, nous n'écrivons pour chaque état que les variables dont la valeur a changé par rapport à l'état précédent.

4.2.4. ST_0

Comme décrit à la section 2.4.4 page 37, un programme ST_0 est un terme. Nous représentons de tels termes de manière arborescente, en les étiquetant par les opérations correspondantes, comme l'illustre la figure 4.5 page 56.

Pour tout terme s, le nœud situé à la racine de s est étiqueté par le symbole $\underline{s}$, qui représente l'opérateur principal du terme s. Par exemple, si s est le terme correspondant au programme $s' \triangleright s''$, avec $s' = \textbf{exit}$ et $s'' = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2$, alors $\underline{s} = \triangleright$, $\underline{s'} = \textbf{exit}$ et $\underline{s''} = \textbf{if}\ e$.

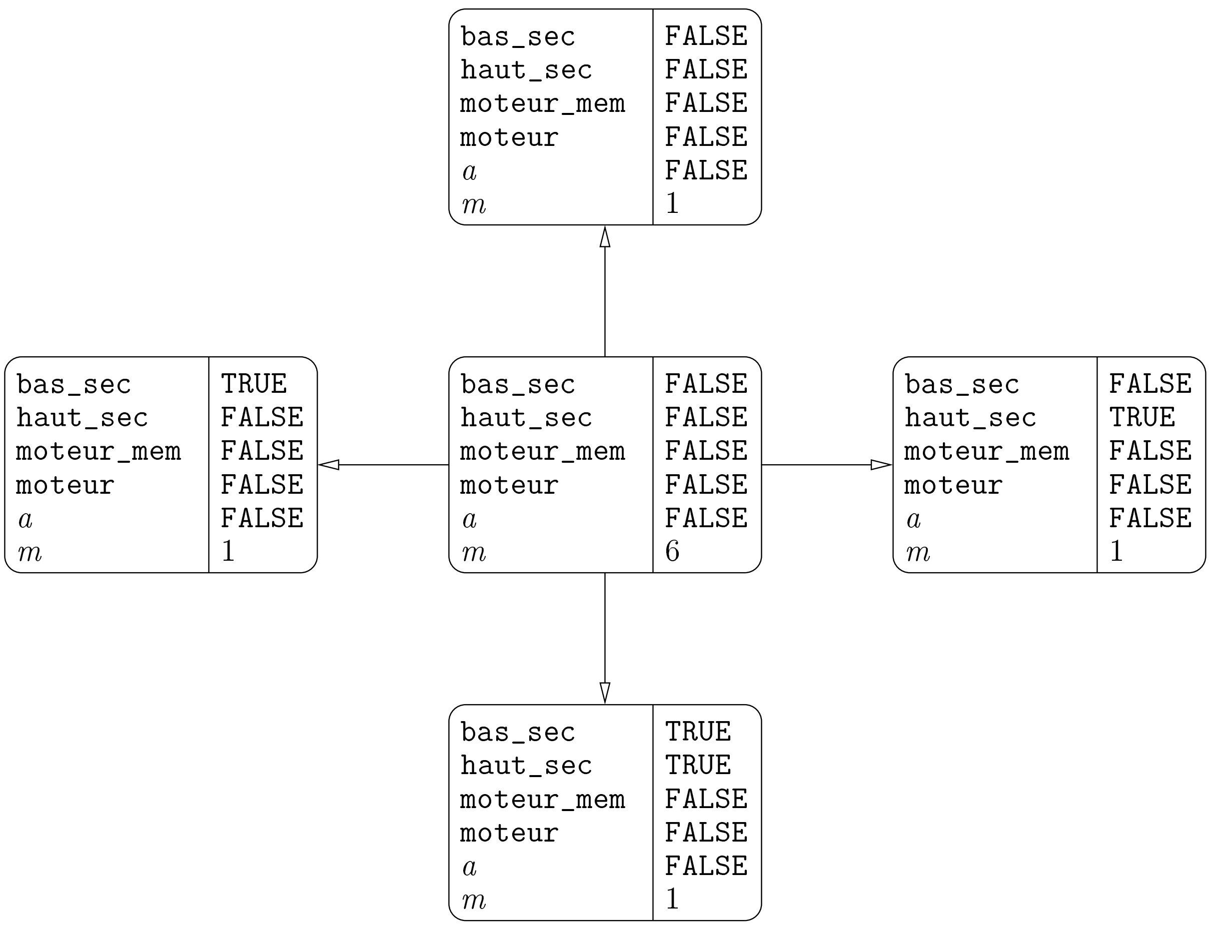

FIG. 4.3.: Transitions d'acquisition à partir d'un état donné.

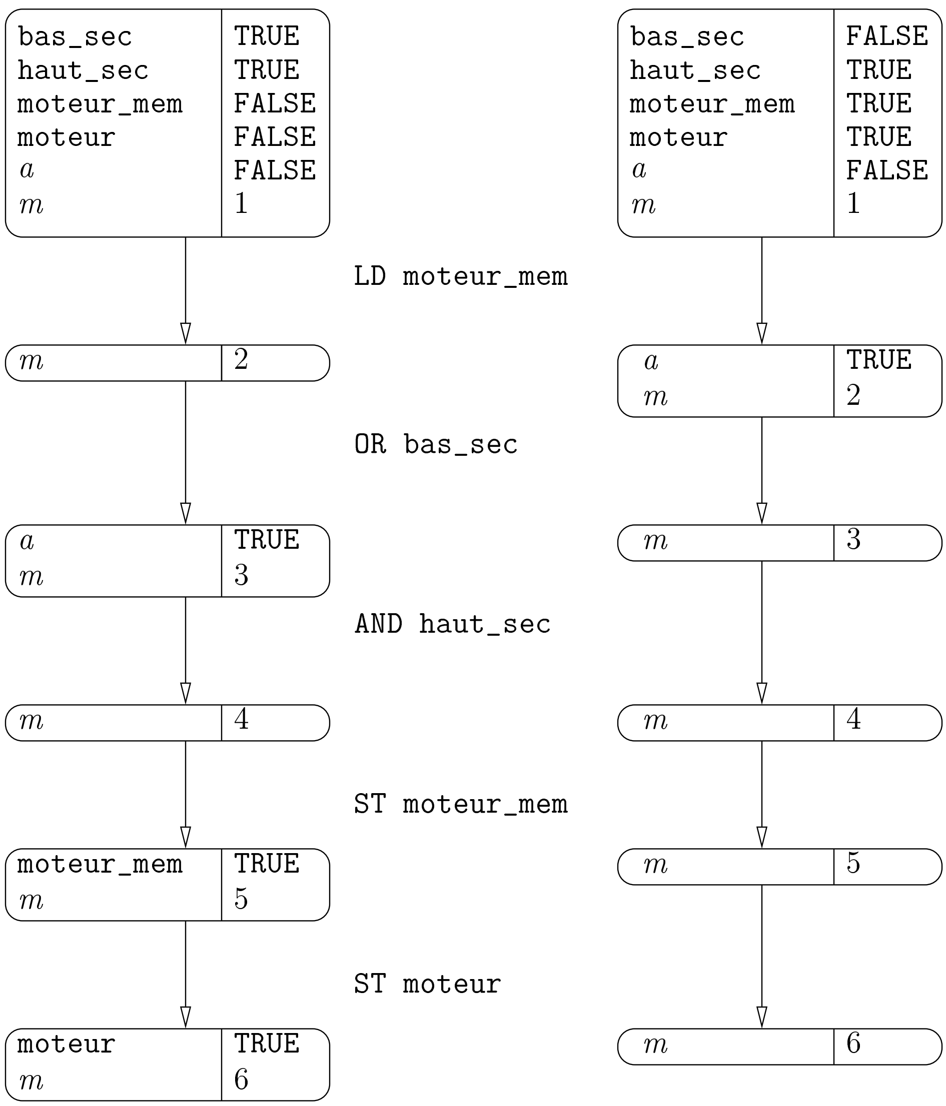

FIG. 4.4.: Déroulement du programme du réservoir à partir de deux situations différentes. Ici, les suites d'instructions sont les mêmes dans les deux cas, mais (en raison des sauts), les instructions exécutées peuvent différer.

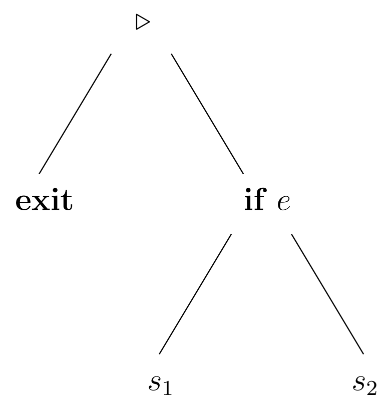

FIG. 4.5.: Terme arborescent étiqueté représentant un programme ST_0.

4.2.4.1. Positions

Afin de représenter la continuation d'un programme (notamment les sauts impliqués par les énoncés **return** et **exit**), nous repérons les positions dans l'arborescence par des mots formés sur l'alphabet $\{1, 2\}$. Par exemple le mot 12 représente le deuxième fils du premier fils de la racine. À ces mots, qui représentent des positions dans un programme, nous ajoutons deux symboles particuliers : \$ représente la terminaison normale et err représente la terminaison par une erreur.

Soit $p \in P_{\mathrm{ST}_0}$ un programme ST_0 bien formé. Nous représentons le codage des positions de p par une fonction $\mapsto$ entre positions et étiquettes, définie de la manière suivante :

si s est un nœud et $w \mapsto \underline{s}$, alors :
- si s_1 est le fils gauche de s, alors $w1 \mapsto \underline{s_1}$,
- si s_2 est le fils droit de s, alors $w2 \mapsto \underline{s_2}$.

De plus, on définit :
- $\$ \mapsto$ **skip** ;
- $err \mapsto$ **skip** ;
- $\varepsilon \mapsto \underline{p}$ (ici, ε représente le mot vide).

Dans la suite, on dira qu'un énoncé s est pointé par w si $w \mapsto \underline{s}$. Nous notons $\mathcal{C}_p$ l'ensemble des positions du programme p.

Exemple Soit le programme $x \leftarrow a \triangleright$ **if** e **then skip else while** e' **do exit**. L'étiquetage du terme correspondant est donné par la figure 4.6 page ci-contre. Les associations suivantes par $\mapsto$ sont définies (et seulement celles-ci) :
- $\$ \mapsto$ **skip** ;
- $err \mapsto$ **skip** ;
- $\varepsilon \mapsto \triangleright$;
- $1 \mapsto x \leftarrow a$;
- $2 \mapsto$ **if** e ;
- $21 \mapsto$ **skip** ;

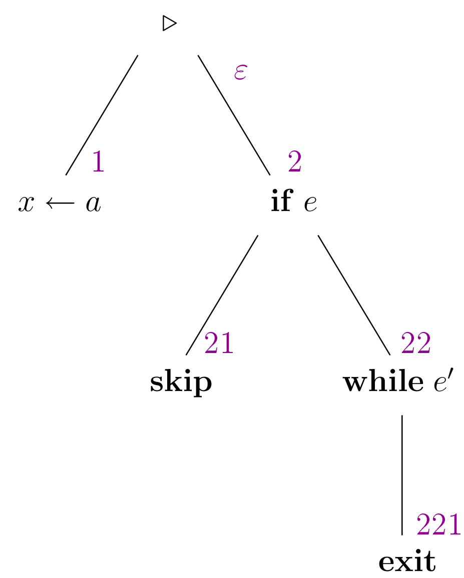

FIG. 4.6.: Terme arborescent étiqueté correspondant au programme $x \leftarrow a \triangleright$ **if** e **then skip else while** e' **do exit**.

- $22 \mapsto$ **while** e' ;
- $221 \mapsto$ **exit**.

Remarque Le nombre d'associations par $\mapsto$ est, de façon évidente, linéaire en la taille de p.

4.2.4.2. Système de transitions

À un programme $p \in P_{\mathrm{ST}_0}$, on associe un système de transitions $\langle Q, \rightarrow, I\rangle$, où :

- l'ensemble des états Q est défini par $Q \stackrel{def}{=} \mathbb{B}^\Sigma \times \mathcal{C}_p$. Un état $q \in Q$ est un couple (V, c), où V est une valuation des variables, et $c \in \mathcal{C}_p$ une position dans le programme ;
- $I \subseteq Q$ est l'ensemble des *états initiaux* : (V, c) est initial si et seulement si $c = \$$ et si, pour toute variable (s, c^i), $V(s) = c^i$;
- la relation de transition $\rightarrow \subseteq Q \times Q$ du système de transitions détaillé associé à un programme p est définie en détail à la section 4.2.4.4 page suivante.

4.2.4.3. Évaluation des expressions

À toute valuation V sur les variables, on associe une fonction $\overline{V}$: à toute expression valide e, la fonction $\overline{V}$ associe une valeur booléenne. $\overline{V}$ est la fonction d'évaluation des expressions et elle est définie comme suit :

- $\overline{V}(\mathtt{TRUE}) = \mathtt{TRUE}$ et $\overline{V}(\mathtt{FALSE}) = \mathtt{FALSE}$;
- $\overline{V}(s) = V(s)$ pour $s \in \Sigma$;
- $\overline{V}(\neg e) = \neg\overline{V}(e)$;

- $\overline{V}(e_1 \vee e_2) = \overline{V}(e_1) \vee \overline{V}(e_2)$;
- $\overline{V}(e_1 \wedge e_2) = \overline{V}(e_1) \wedge \overline{V}(e_2)$.

Ainsi par exemple, si V est telle que $V(a) = \texttt{TRUE}$ et $V(b) = \texttt{FALSE}$, alors $\overline{V}(a \vee b) = \texttt{TRUE}$.

4.2.4.4. Définition des transitions

Cette section définit exhaustivement l'ensemble des transitions d'un système de transitions associé à un programme $p \in \mathrm{ST}_0$.

Fonctions d'aller et retour Nous utiliserons dans cette définition deux fonctions aller et retour, qui effectuent des correspondances sur les positions :
- la fonction aller détermine le prochain énoncé à exécuter si l'énoncé pointé est une séquence ($\triangleright$) ;
- la fonction retour détermine l'énoncé suivant à exécuter si l'énoncé pointé est terminal (situé sur une feuille de l'arbre ou en fin d'exécution de boucle **while** : pas de continuation possible à partir de cet énoncé). La définition du retour est dépendante d'un mode, *continue* ou *exit*, qui donne une indication sur la raison du retour (la destination du retour n'est pas la même si on est en train d'exécuter un **exit**).

Exemple Nous avons, pour le programme de la figure 4.6 page précédente :
- $\mathrm{aller}(\varepsilon) = 1$ car l'énoncé $x \leftarrow a$ est le premier sous-énoncé de la séquence à devoir être exécuté ;
- $\mathrm{aller}(2) = 2$ car l'énoncé pointé par la position 2 n'est pas une sélection (c'est **if** e) ;
- $\mathrm{retour}(21, \mathit{continue}) = \$$ car, à la fin de l'exécution de **skip**, l'exécution du programme est terminée ;
- $\mathrm{retour}(221, \mathit{exit}) = \$$ car l'énoncé **exit** provoque l'arrêt de la boucle **while** e'.

Soit $\mathit{Mode} = \{\mathit{continue}, \mathit{exit}\}$ un ensemble de deux symboles. On définit les fonctions aller : $\mathcal{C}_p \setminus \{\$, \mathit{err}\} \to \mathcal{C}_p$ et retour : $(\mathcal{C}_p \setminus \{\$, \mathit{err}\}) \times \mathit{Mode} \to \mathcal{C}_p$.

Définition de la fonction aller Pour tout $w \in \mathcal{C}_p$, on définit $\mathrm{aller}(w)$ de la façon suivante :
- $\mathrm{aller}(w) = \mathrm{aller}(w1)$ si $w \mapsto \triangleright$;
- $\mathrm{aller}(w) = w$ sinon.

Définition de la fonction retour Dans le cas où $w = \varepsilon$, on définit :
- $\mathrm{retour}(\varepsilon, \mathit{continue}) = \$$ (c'est une terminaison normale) ;
- $\mathrm{retour}(\varepsilon, \mathit{exit}) = \mathit{err}$ (cas d'erreur : un **exit** n'est pas sous-terme d'un **while**).

Si $w \neq \varepsilon$, alors on isole le dernier chiffre de w et on définit par cas en fonction de ce chiffre : on définit $v \in \mathcal{C}_p$ et $a \in \{1, 2\}$ tels que $w = va$. On définit alors $\mathrm{retour}(w)$ comme suit :

1. cas $v \mapsto$ **if** e :

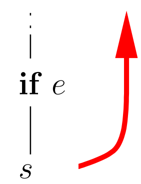

FIG. 4.7.: Définition de la fonction retour à partir d'un énoncé **if**.

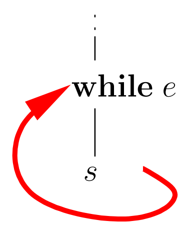

FIG. 4.8.: Définition de la fonction retour à partir d'un énoncé **while** sans avoir rencontré **exit**.

- retour(va, x) = retour(v, x) pour tous $x \in \mathit{Mode}$ et $a \in \{1,2\}$[2] (fig. 4.7 page suivante) ;

2. cas $v \mapsto$ **while** e :
   - retour$(va, \mathit{continue}) = v$ pour tout $a \in \{1,2\}$[3] (fig. 4.8),
   - retour(va, exit) = retour$(v, \mathit{continue})$ pour tout $a \in \{1,2\}$[4] (fig. 4.9) ;
3. cas $v \mapsto \triangleright$:
   - retour$(v2, x)$ = retour(v, x) pour tout $x \in \mathit{Mode}$[5] (fig. 4.10),
   - retour$(v1, \mathit{continue})$ = aller$(v2)$[6] (fig. 4.11) ;
   - retour$(v1, \mathit{exit})$ = retour(v, exit)[7] (fig. 4.12).

[2] Fin d'exécution d'une clause **then** ou **else** qui entraîne un retour. On saute le **if** et on effectue récursivement le retour au-dessus.
[3] Fin de l'énoncé du **while**, aucun **exit** n'a été lu, nouvelle itération de la boucle.
[4] Fin de l'énoncé du **while**, un **exit** a été lu, on effectue récursivement le retour au-dessus du **while** en absorbant le **exit** – retour en mode *continue*.
[5] Fin d'exécution du deuxième énoncé d'une séquence, on remonte au-dessus de la séquence.
[6] Fin d'exécution du premier énoncé d'une séquence sans avoir lu de **exit**, on passe au deuxième énoncé.
[7] Fin d'exécution du premier énoncé d'une séquence en ayant lu un **exit**, on arrête l'exécution de la

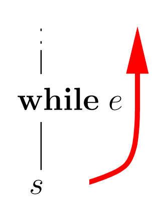

FIG. 4.9.: Définition de la fonction retour à partir d'un énoncé **while** en ayant rencontré **exit**.

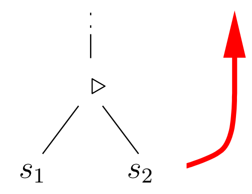

Fig. 4.10.: Définition de la fonction retour à partir du deuxième énoncé d'une séquence.

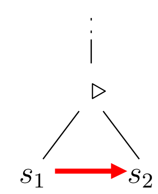

Fig. 4.11.: Définition de la fonction retour à partir du premier énoncé d'une séquence, sans avoir rencontré **exit**.

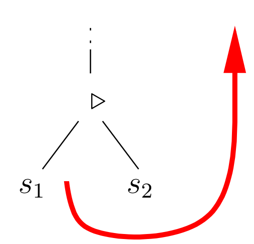

Fig. 4.12.: Définition de la fonction retour à partir du premier énoncé d'une séquence, en ayant rencontré **exit**.

Pour tout $w \notin \{\$, err\}$ tel que $w \mapsto$	il existe une transition $(V, w) \rightarrow$
$x \leftarrow e$	$(V\{x/\overline{V}(e)\}, \text{retour}(w, \mathit{continue})$
return	$(V, \$)$
skip	$(V, \text{retour}(w, \mathit{continue}))$
if e	$(V, \text{aller}(w1))$ si $\overline{V}(e) = \texttt{TRUE}$ $(V, \text{aller}(w2))$ sinon
while e	$(V, \text{aller}(w1))$ si $\overline{V}(e) = \texttt{TRUE}$ $(V, \text{retour}(w, \mathit{continue}))$ sinon
exit	$(V, \text{retour}(w, \mathit{exit}))$

TAB. 4.2.: Relation de transition d'un système représentant un programme ST_0.

Relation de transition Dans le cas où $w \notin \{\$, err\}$, les transitions du système sont données par le tableau 4.2.

Lecture des entrées Comme pour IL_0, nous ajoutons des transitions représentant l'acquisition de entrées : pour toutes valuations V et V' ne différant que sur leurs variables d'entrée, on crée une transition de $(V, \$)$ vers $(V', \text{aller}(\varepsilon))$.

Erreur L'état d'erreur est bloquant :

$$(V, err) \rightarrow (V, err) \text{ pour tout } V$$

4.2.4.5. Déterminisme

De même que pour IL_0, on remarque que seules les transitions d'acquisition des entrées sont non-déterministes.

4.2.4.6. Exemple

Nous prenons l'exemple d'un programme contenant deux boucles **while** et un **exit** conditionnel. Nous le représentons figure 4.13 sous la forme d'un pseudo-programme (nous ne déclarons pas les variables et ne détaillons pas leur utilisation ; c'est seulement la structure du programme qui nous intéresse).

Ce programme s'écrit, dans la syntaxe abstraite

$$x \leftarrow a \triangleright \textbf{while } e \textbf{ do while } e' \textbf{ do if } e'' \textbf{ then exit else skip} \triangleright y \leftarrow b$$

La représentation arborescente étiquetée en est donnée par la figure 4.14. Le tableau 4.3 montre la définition des positions du programme. Le tableau 4.4 donne l'application de la fonction retour aux positions du programme en fonction du mode.

L'application de la fonction aller donne les valeurs suivantes :

séquence et on remonte au-dessus.

```
x := a;
WHILE e DO
  WHILE e' DO
    IF e''
    THEN EXIT;
    END_IF;
  END_WHILE;
END_WHILE;
y := b;
```

Fig. 4.13.: Pseudo-programme ST.

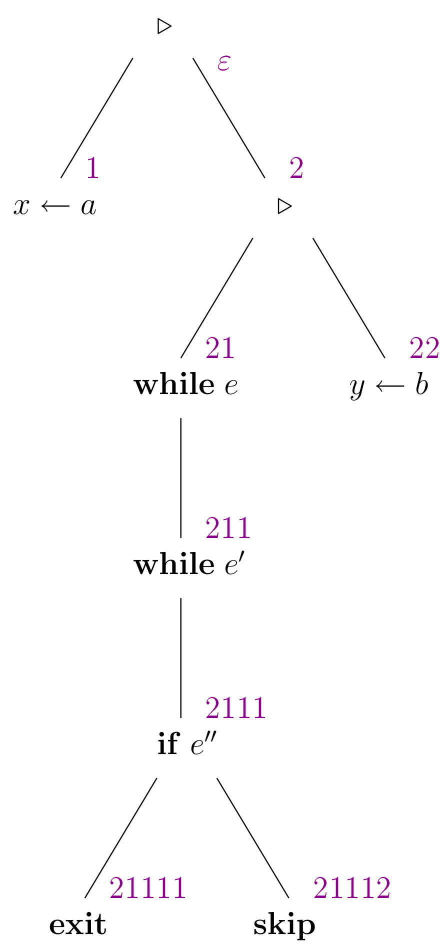

Fig. 4.14.: Représentation arborescente du programme de l'exemple.

Positions	$\mapsto$
\$	**skip**
err	**skip**
ε	$\triangleright$
1	$x \leftarrow a$
2	$\triangleright$
21	**while** e
22	$x \leftarrow b$
211	**while** e'
2111	**if** e''
21111	**exit**
21112	**skip**

TAB. 4.3.: Définition des positions pour l'exemple.

	$continue$	$exit$
ε	\$	err
1	21	err
2	\$	err
21	22	err
22	\$	err
211	21	22
2111	211	21
21111	211	21
21112	211	21

TAB. 4.4.: Définition de la fonction retour sur l'exemple.

- aller(ε) = 1 ;
- aller(2) = 21 ;
- aller(w) = w pour tout $w \notin \{\varepsilon, 2\}$.

En faisant abstraction des valuations des variables, les transitions du système donnent les liaisons suivantes entre continuations :

- $\$ \to 1$;
- $1 \to 21$;
- $21 \to 211$ si $V(e) = \texttt{TRUE}$;
- $21 \to 22$ si $V(e) = \texttt{FALSE}$;
- $22 \to \$$;
- $211 \to 2111$ si $V(e') = \texttt{TRUE}$;
- $211 \to 21$ si $V(e') = \texttt{FALSE}$;
- $2111 \to 21111$ si $V(e'') = \texttt{TRUE}$;
- $2111 \to 21112$ si $V(e'') = \texttt{FALSE}$;
- $21111 \to 21$;
- $21112 \to 211$.

Dans cet exemple, l'état d'erreur n'est pas accessible (le **exit** est bien dans un **while**).

4.2.5. Conclusion

Il est possible de décrire totalement la sémantique des langages IL_0 et ST_0 à l'aide d'un formalisme aussi simple que celui des systèmes de transitions non étiquetés présentés dans cette section. Le système de transitions associé à un programme peut être codé dans le langage d'entrée d'un model-checker, de manière assez directe (voir les sections 6.2 page 119 et 6.3 page 121), et surtout, comme le montrent les exemples des sections précédentes, de manière compacte. On verra que le code produit à partir de cette sémantique est efficace du point de vue de la vérification.

Cette sémantique a cependant le défaut de ne pas être modulaire : un programme se considère dans sa globalité. Cela n'est pas gênant pour un langage tel que IL, qui est très faiblement structuré : la sémantique de IL, ainsi exprimée, est très simple et naturelle ; cela permet aussi bien de la comprendre que de l'étendre facilement (voir la discussion sur l'extension des sémantiques, section 4.4 page 88).

Le cas de ST est plus délicat. La définition (et la compréhension) de la sémantique de ST, dans ce formalisme, est pénible et peu naturelle. La principale conséquence d'une telle définition est qu'elle pose un problème de validation. En effet, contrairement au cas du IL, il n'est pas évident de se convaincre du fait qu'un modèle corresponde bien au programme qu'il représente. De plus, elle se prête peu à d'éventuelles extensions. Puisque le but de notre projet est de définir une sémantique de ST qui s'intègre naturellement dans un cadre multilangage, il est nécessaire de prendre en compte, directement dans le formalisme utilisé, la modularité du langage.

4.3. Sémantique modulaire

4.3.1. Introduction

Cette section présente la sémantique modulaire, telle qu'elle a été adoptée dans le projet Vulcain. Contrairement au formalisme décrit précédemment, celui-ci est explicitement conçu pour définir une sémantique unifiée à trois des langages de la norme (dont ST). Nous décrivons ici le formalisme utilisé, et la sémantique du langage ST.

Certains choix effectués lors de la définition du formalisme dépendent de l'aspect multilangage des programmes, et du cas très particulier du langage SFC, où ils trouvent leur justification. Nous ne décrirons pas tous ces aspects ici ; nous nous bornerons à présenter la sémantique telle qu'elle a été adoptée. Cette sémantique a l'avantage de permettre une définition simple et compréhensible du comportement des programmes ST, en offrant une prise en compte naturelle de la modularité des langages.

4.3.2. Définition du formalisme

4.3.2.1. Format des automates utilisés

Nous utilisons, pour définir la sémantique opérationnelle de tout le langage défini par la norme, des automates synchronisés par envois de messages. Il s'agit d'une classe particulière de systèmes de transitions étiquetés. On associe à chaque élément de langage (expression ou énoncé) un automate qui représente son comportement ; cela permet de définir la sémantique des langages de façon modulaire, élément par élément. Les étiquettes sur les transitions permettent de définir les interactions entre les automates (qui représentent les interactions entre les éléments de langage représentés par ces automates). Par exemple, considérons l'énoncé ST suivant :

```
x := y ;
```

L'énoncé est l'affectation d'une variable dans une autre. Pour représenter le comportement décrit par cet énoncé, trois automates sont utilisés : un pour représenter la variable x, un autre pour la variable y, un troisième pour l'opération d'affectation. Les trois automates communiquent entre eux par échange de messages, comme l'illustre la figure 4.15.

Le formalisme des automates synchronisés par envois de messages, particulièrement adapté à la description de telles interactions, est utilisé pour représenter les éléments de langage de la norme. La synchronisation est une forme de communication entre automates, sur le principe du rendez-vous : si deux automates doivent se synchroniser sur une transition précise, ils ne pourront franchir cette transition que simultanément. Cette méthode permet de représenter, par exemple, un appel de fonction, et la mise en attente de l'appelant sur le résultat. Comme décrit dans les sections suivantes, toutes les formes de langage qui font l'objet de ce rapport peuvent être représentées par ce formalisme, de manière très modulaire.

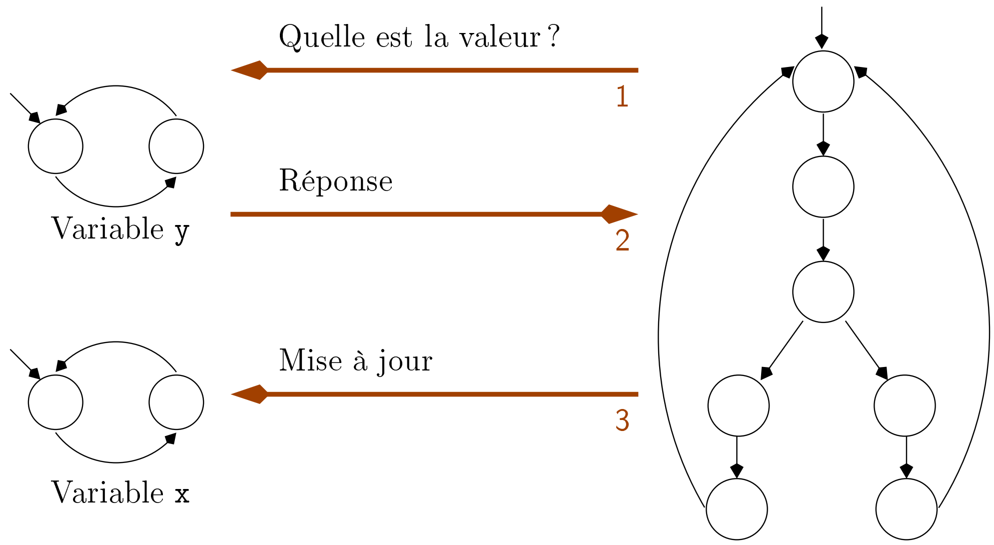

Fig. 4.15.: Illustration des communications entre automates.

	Rendez-vous	Broadcast
Émission	$!m$	$!!m$
Réception	$?m$	$??m$

Tab. 4.5.: Description de la nature des messages de synchronisation.

4.3.2.2. Propositions

Dans le but d'exprimer des propriétés sur le système, nous définissons un étiquetage des états. Les états du système sont ainsi étiquetés par des symboles, que nous appelons propositions. Les propositions permettent de distinguer certains états en fonction de propriétés intrinsèques que l'on souhaite mettre en évidence. Il s'agira principalement de propositions sur les valeurs des variables, et sur le contrôle (fin de cycle, erreur).

4.3.2.3. Messages

Les symboles de messages sont donnés par un ensemble M. Les messages peuvent être émis ou reçus. Ces émissions et réceptions sont notées comme indiqué dans le tableau 4.5, où m désigne un élément de M.

Il existe également un cinquième type de message, les messages m sans émission ni réception, que nous nommerons les messages *neutres*. Par la suite nous désignerons par Σ_M l'ensemble des messages et des émissions et réceptions associées :

$$\Sigma_M = \{m, !m, ?m, !!m, ??m / m \in M\}$$

Il n'est pas classique de différencier les messages $!m$ et $!!m$ (nous appellerons ces messages respectivement *rendez-vous* et *broadcast*). Les messages jouent un rôle légèrement différent lors de la synchronisation, comme cela est expliqué à la page 70.

$\bullet$	$!m$	$!!m$	$?m$	$??m$	m
$!m$			m	$??m$	
$!!m$			$!!m$		
$?m$	m	$!!m$			
$??m$	$??m$				
m					

TAB. 4.6.: Définition de l'opérateur $\bullet$.

4.3.2.4. Synchronisation de messages

Les émissions et réceptions de messages peuvent bien entendu se synchroniser. Pour modéliser cela, nous définissons un opérateur commutatif $\bullet$ de $\Sigma_M \times \Sigma_M$ vers Σ_M. Cet opérateur, partiellement défini, est donné par le tableau 4.6. Une case blanche signifie que l'opérateur n'est pas défini. En particulier, l'opérateur $\bullet$ n'est jamais défini pour deux messages de valeurs différentes.

Ainsi une émission peut se synchroniser avec une réception, sauf dans le cas où les deux sont des messages de broadcast. Dans tous les autres cas la synchronisation n'est pas autorisée.

4.3.2.5. Automates représentant la sémantique opérationnelle des éléments de langage

Un automate représentant un élément de langage est formellement représenté par un sextuplet $A = \langle M, \Pi, Q, q^I, \rightarrow, l \rangle$, où :

- M est un ensemble de symboles (les messages) ;
- Π est un ensemble de symboles (les propositions) ;
- Q est l'ensemble des états de l'automate ;
- q^I est l'état initial de l'automate ($q^I \in Q$) ;
- $\rightarrow \in Q \times \Sigma_M \times Q$ est la relation de transition ;
- l est la fonction d'étiquetage des états ($l : Q \rightarrow 2^\Pi$).

L'ensemble de symboles Π est l'ensemble des propositions atomiques pouvant être placées en étiquettes sur les états. Une transition est un triplet (q, α, q'), signifiant que A peut passer de l'état q à l'état q' par l'action α. Pour tout triplet $(q, \alpha, q') \in \rightarrow$, on note $q \xrightarrow{\alpha} q'$.

À tout état $q \in Q$, la fonction l associe un sous-ensemble de Π ; par exemple $l(q) = \{P_1, \ldots, P_n\}$ signifie que q est un état qui vérifie $P_1, \ldots, P_n$ et qui ne vérifie aucune des autres propositions de Π. Comme c'est l'usage, un automate sera le plus souvent représenté par un diagramme. Par exemple la figure 4.16 représente un automate dans lequel :

- $M \supseteq \{i, o, mn\}$ (il peut y avoir d'autres éléments) ;
- $\Pi \supseteq \{a, b, c\}$ (id.) ;
- $Q = \{q_0, q_1\}$;
- $q^I = q_0$;
- $\rightarrow = \{(q_0, ?i, q_1), (q_1, !o, q_0), (q_1, !mn, q_1)\}$;

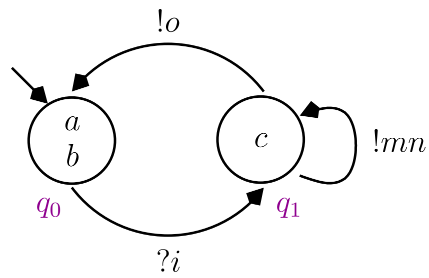

FIG. 4.16.: Exemple d'automate étiqueté.

– $l(q_0) = \{a, b\}$, $l(q_1) = \{c\}$.

Il est bien sûr possible, comme ce sera souvent le cas, d'avoir des états q sans étiquette (alors $l(q) = \emptyset$).

L'état initial est signalé par une flèche sans origine. Dans la suite, nous ne ferons pas apparaître les noms des états car ils sont en fait sans importance réelle.

Interface L'interface d'un automate A, notée $\mathrm{I}(A)$, est constituée de toutes les actions apparaissant dans les transitions de A. Formellement

$$\mathrm{I}(A) = \{\alpha \in \Sigma_M \setminus M / \exists q, q', q \xrightarrow{\alpha} q' \text{ est une transition de } A\}$$

4.3.2.6. Synchronisation d'automates

Nous utilisons la synchronisation entre automates pour combiner les automates entre eux, et construire les automates associés aux programmes. Le principe de la synchronisation entre deux automates est de rassembler les deux automates pour n'en faire qu'un (sur le principe d'un produit d'automates), en prenant en compte les messages de synchronisation entre les deux automates. Cela suppose notamment une définition de la relation de transition :

1. on ajoute les transitions résultant d'une synchronisation (déterminées par l'opérateur $\bullet$) ;
2. on supprime les transitions où la synchronisation n'a pas pu se faire. Ces modifications concernent les cas où une transition du premier automate peut être synchronisée avec une transition du deuxième (le critère est l'opérateur de synchronisation appliqué aux étiquettes des transitions). Dans ce cas, seules sont conservées les transitions issues de la synchronisation entre ces deux transitions. Toutes les transitions issues d'une seule de ces deux transitions sont supprimées. En dehors de ces deux cas, les transitions existantes sont conservées.

La figure 4.17 illustre la synchronisation entre deux automates. On voit que, par exemple, les transitions ayant pour origine les états q_0 et s_0 ne peuvent être franchies que simultanément. Les étiquettes de transitions $!i$ et $?i$ sont alors remplacées par une étiquette neutre i (car $?i \bullet !i = i$). La transition $!m$, par contre, a été conservée, car il

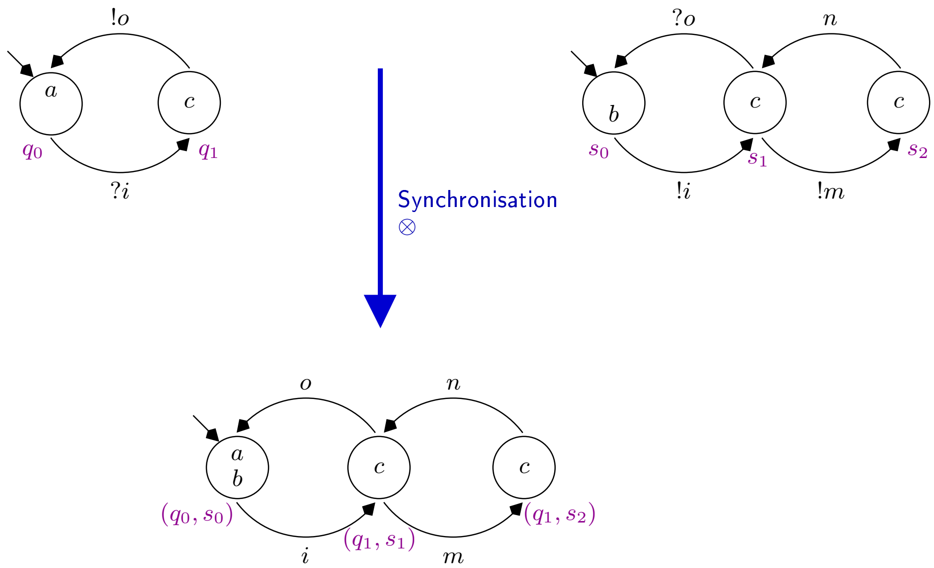

FIG. 4.17.: Synchronisation entre deux automates.

n'existe dans le premier automate aucune transition pouvant être synchronisée avec celle-ci (c'est-à-dire telle que $\alpha \bullet !m$ soit défini, quel que soit α étiquette de transition du premier automate). Il en est de même pour la transition portant le message neutre n (l'opérateur de synchronisation entre messages, décrit par le tableau 4.6 page 67, n'est jamais défini pour les messages neutres).

Définition formelle de la synchronisation entre automates Soient A_1 et A_2 ($A_i = \langle M_i, \Pi_i, Q_i, q_i^I, \rightarrow_i, l_i \rangle$) deux automates tels que $\Pi_1 \cap \Pi_2 = \emptyset$. L'automate A, synchronisation de A_1 et A_2, noté $A = A_1 \otimes A_2$, est défini tel que $A = \langle M_1 \cup M_2, \Pi_1 \cup \Pi_2, Q_1 \times Q_2, (q_1^I, q_2^I), \rightarrow, l \rangle$. La relation de transition $\rightarrow$ et la fonction d'étiquetage des états l sont décrites ci-dessous.

Relation de transition Pour tous $(q_1, q_1') \in Q_1^2$, $(q_2, q_2') \in Q_2^2$, on définit une transition $(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2')$ si :

- $\exists \alpha_1, \alpha_2, q_1 \xrightarrow{\alpha_1}_1 q_1'$, $q_2 \xrightarrow{\alpha_2}_2 q_2'$ avec $\alpha = \alpha_1 \bullet \alpha_2$. C'est une transition synchronisée ;
- $q_1 \xrightarrow{\alpha}_1 q_1'$, $q_2 = q_2'$, dans l'un des cas suivants :
  - ou bien α est un message de broadcast ou un message neutre,
  - ou bien α est un message de rendez-vous et il n'existe pas de $\alpha_2 \in \mathrm{I}(A_2)$ tel que $\alpha \bullet \alpha_2$ est défini ;
- symétriquement, $q_2 \xrightarrow{\alpha}_2 q_2'$, $q_1 = q_1'$, dans l'un des cas suivants :
  - ou bien α est un message de broadcast ou un message neutre,
  - ou bien α est un message de rendez-vous et il n'existe pas de $\alpha_1 \in \mathrm{I}(A_1)$ tel que $\alpha_1 \bullet \alpha$ est défini.

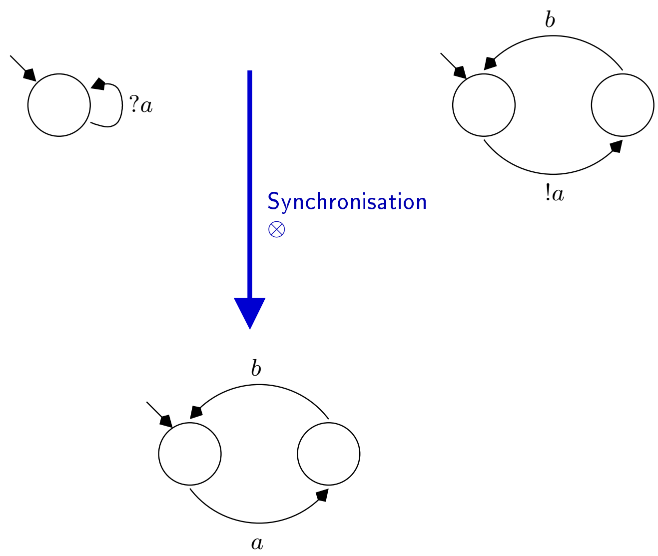

Fig. 4.18.: Exemple de synchronisation avec une transition de rendez-vous.

Différence entre rendez-vous et broadcast La différence entre un message de rendez-vous et un message de broadcast est que les transitions dont l'étiquette est un message de broadcast sont conservées, même si la synchronisation est possible. Alors que les transitions de rendez-vous, si la synchronisation est possible, ne doivent être franchies que deux à deux simultanément, les transitions de broadcast peuvent aussi bien être franchies simultanément que séparément (elles ne sont donc pas supprimées et peuvent être réutilisées pour la synchronisation avec un autre automate). Par exemple, la figure 4.18 montre la synchronisation entre deux transitions de rendez-vous ; la figure 4.19 montre la synchronisation entre une transition de rendez-vous et une transition de broadcast.

Étiquetage des états La fonction $l : Q_1 \times Q_2 \rightarrow \Pi_1 \cup \Pi_2$ d'étiquetage des états est définie par $l((q_1, q_2)) = l_1(q_1) \cup l_2(q_2)$.

4.3.2.7. Associativité et ordre de construction

Dans le cas général, la synchronisation définie à la section 4.3.2.6 page 68 n'est pas associative. Par exemple, la figure 4.20 illustre trois automates A_1, A_2, A_3 et la synchronisation $(A_1 \otimes A_2) \otimes A_3$. La figure 4.21 montre la synchronisation $A_1 \otimes (A_2 \otimes A_3)$ sur les mêmes automates.

Il convient donc, lors de la synchronisation de plusieurs automates entre eux, de définir parfaitement l'ordre dans lequel la synchronisation est effectuée.

4.3.2.8. Automates indépendants

Il existe malgré tout un cas où le produit de synchronisation de plusieurs automates est indépendant de l'ordre. C'est le cas où les automates sont mutuellement indépendants.

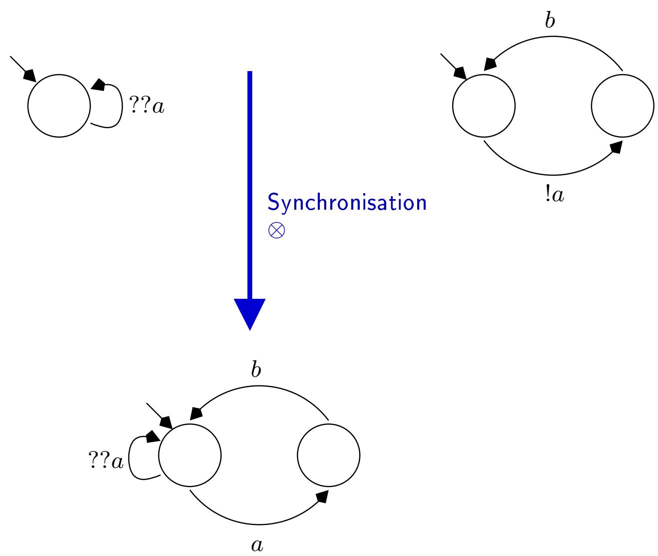

FIG. 4.19.: Exemple de synchronisation avec une transition de broadcast.

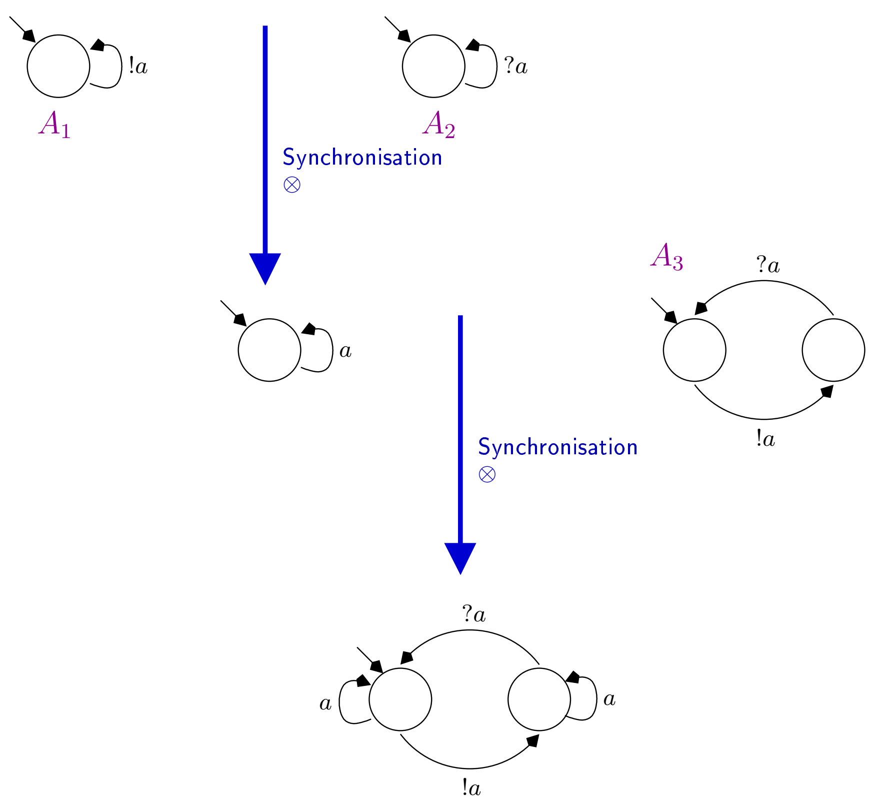

FIG. 4.20.: Synchronisation entre trois automates dans un ordre donné.

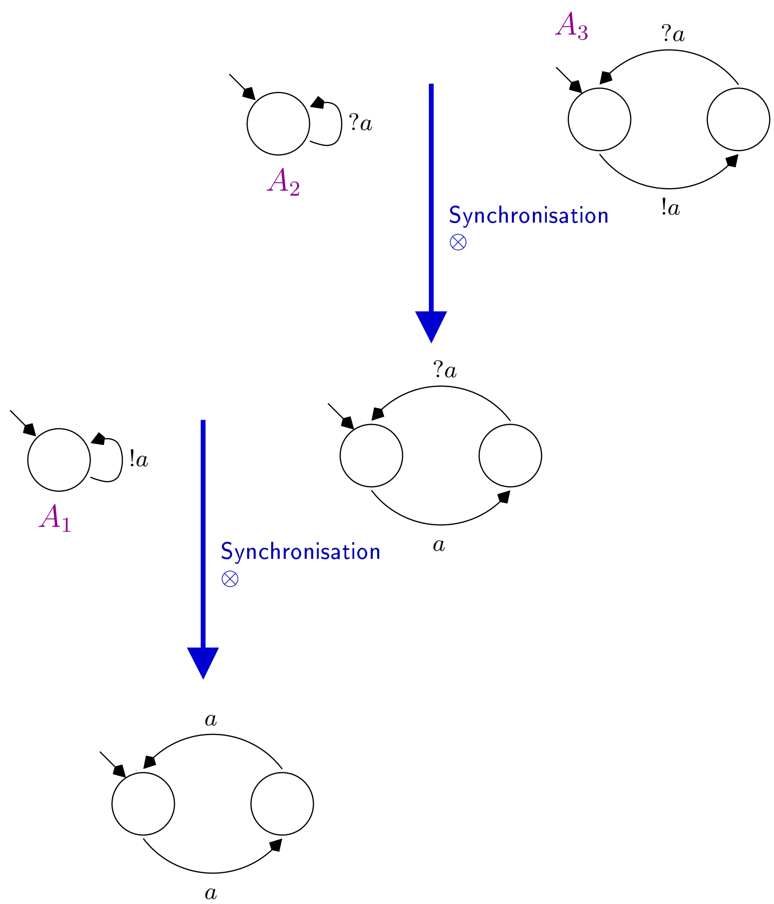

FIG. 4.21.: Synchronisation entre les trois mêmes automates qu'en 4.20 dans un ordre différent.

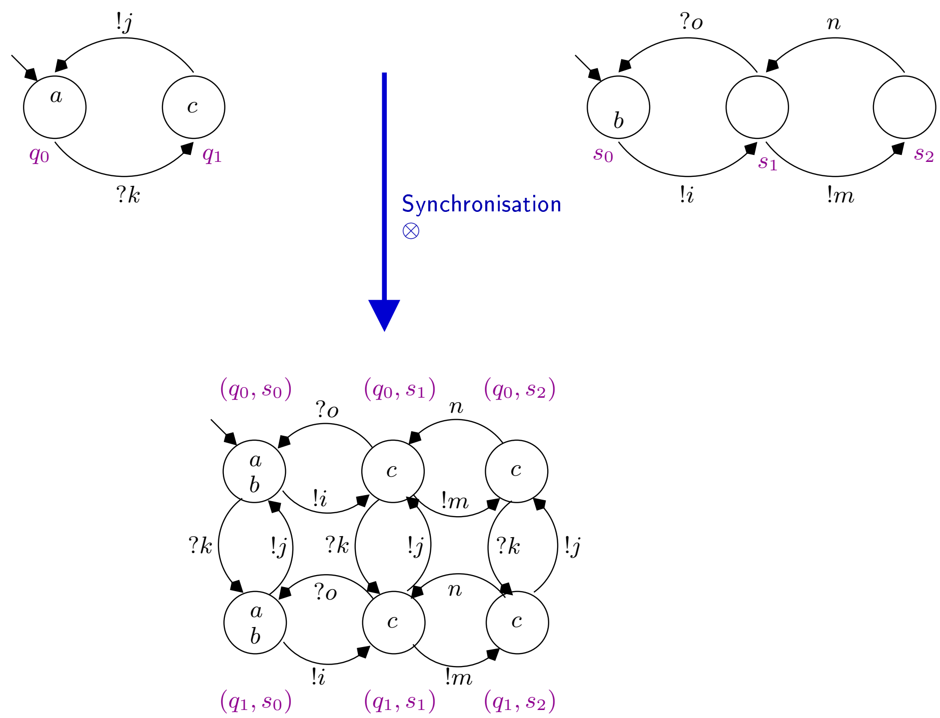

Fig. 4.22.: Synchronisation entre deux automates indépendants.

On dit que deux automates A_1 et A_2 sont mutuellement dépendants s'il existe un couple $(\alpha_1, \alpha_2) \in \mathrm{I}(A_1) \times \mathrm{I}(A_2)$ tel que $\alpha_1 \bullet \alpha_2$ est défini. Inversement, s'il n'existe pas de tel couple, on dit que A_1 et A_2 sont mutuellement indépendants. Si deux automates sont mutuellement indépendants, la synchronisation revient à faire le produit des deux automates, comme l'illustre la figure 4.22. Aucune transition n'est supprimée. Dans le cas où trois automates A_1, A_2 et A_3 sont mutuellement indépendants deux à deux, on a aussi $A_1 \otimes A_2$ indépendant de A_3. On en déduit $(A_1 \otimes A_2) \otimes A_3 = A_1 \otimes (A_2 \otimes A_3)$.

4.3.2.9. Renommage des étiquettes de transitions

Notre modèle permet de réutiliser plusieurs instances du même automate lors d'une opération de synchronisation. Prenons l'exemple d'un automate devant communiquer avec deux automates représentant le même élément de langage. Il est nécessaire de pouvoir différencier les messages adressés au premier des messages adressés au second. Cette opération s'effectue en renommant les étiquettes des transitions. La figure 4.23 montre le renommage de deux automates A_1 et A_2 de même nature. Ces deux automates ont un rôle différent vis-à-vis de la synchronisation avec l'automate A. Bien que différents, ces automates partagent une même structure, et on peut donc les obtenir de façon uniforme à partir d'un schéma unique, en utilisant le renommage.

Soient M et M' deux ensembles de messages et c une application de M vers M'. Pour tout $\alpha = !m$ (resp. $?m$, $!!m$, $??m$, m) de Σ_M, on note $c(\alpha) \stackrel{def}{=} !c(m)$ (resp. $?c(m)$, $!!c(m)$,

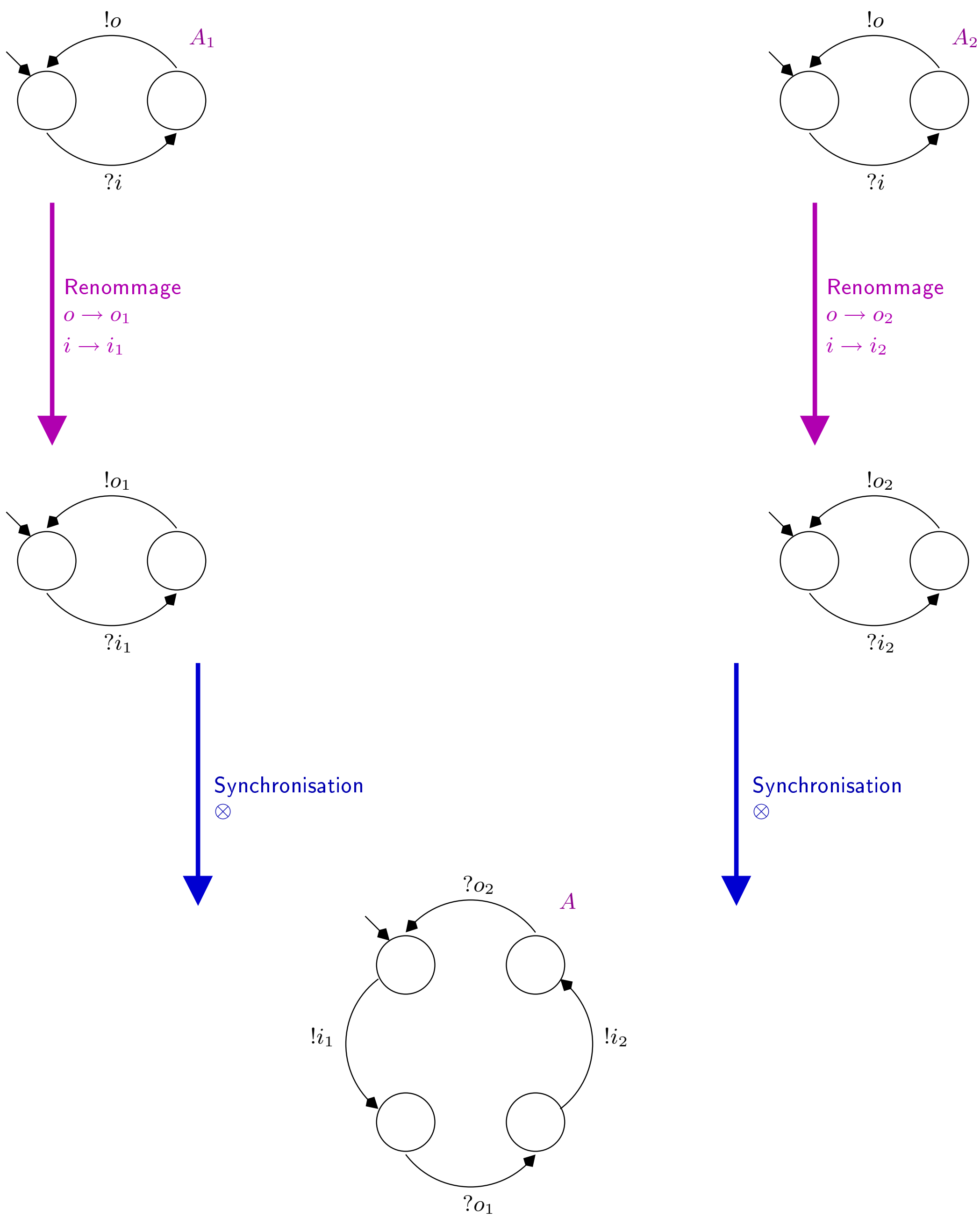

Fig. 4.23.: Renommage d'automates.

??$c(m)$, $c(m)$) l'élément de $\Sigma_{M'}$ correspondant.

Si $A = \langle M, \Pi, Q, q^I, \rightarrow, l \rangle$ est un automate, on définit l'automate renommé de A par c comme l'automate $\langle M', \Pi, Q, q^I, \rightarrow', l \rangle$ avec, pour tous $(q, q') \in Q^2$, $q \xrightarrow{\beta}{}' q'$ s'il existe un α tel que $\beta = c(\alpha)$ et $q \xrightarrow{\alpha} q'$.

Notation Si, pour tout $a_i \in M$, on a $c(a_i) = a'_i$, le renommage de A par c est noté $A\{a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n\}$ (il n'est pas nécessaire de noter les éléments de M qui conservent leur valeur).

4.3.2.10. Renommage des étiquettes d'états

On définit de manière similaire un renommage des étiquettes d'états : soient Π et Π' deux ensemble de propositions et d une application de Π vers Π'.

Si $A = \langle M, \Pi, Q, q^I, \rightarrow, l \rangle$ est un automate, on définit l'automate renommé de A par c comme l'automate $\langle M, \Pi', Q, q^I, \rightarrow, l' \rangle$ avec, pour tous $(q, q') \in Q^2$, $l'(q') = \{d(p)/p \in l(q)\}$.

Notation Si, pour tout $p_i \in \Pi$, on a $d(p_i) = p'_i$, le renommage de A par d est noté $A\{p_1 \Rightarrow p'_1, \ldots, p_n \Rightarrow p'_n\}$.

4.3.2.11. Restriction d'un automate par un ensemble de messages

Il existe également une autre opération sur les automates, utilisée pour les programmes multilangages, mais que nous n'utiliserons pas dans le cadre des programmes écrits dans un seul langage. L'opération de restriction permet de supprimer des transitions d'un automate en fonction des messages qu'elles portent.

Soit un automate A tel que $A = \langle M, \Pi, Q, q^I, \rightarrow, l \rangle$, soit $X \subseteq \Sigma_M$ un ensemble de messages. On définit l'automate restreint de A par X comme l'automate $\langle M, \Pi, Q, q^I, \rightarrow', l \rangle$, avec, pour tous $(q, q') \in Q^2$, $q \xrightarrow{\alpha}{}' q'$ si $q \xrightarrow{\alpha} q'$ et $\alpha \notin X$.

Notation Si $X = \{\alpha_1, \ldots, \alpha_n\}$ la restriction de A par X est notée $A - \{\alpha_1, \ldots, \alpha_n\}$.

4.3.3. Sémantique modulaire ST

4.3.3.1. Variables

La figure 4.24 montre l'automate A_x, représentant une variable. Une variable peut :
- envoyer des messages indiquant sa valeur courante ;
- recevoir des messages d'affectation pour modifier (ou non) sa valeur.

Dans le cas de A_x, on a $M = \{x \leftarrow \texttt{FALSE}, x \leftarrow \texttt{TRUE}, x = \texttt{FALSE}, x = \texttt{TRUE}\}$ et $\Pi = \{x, \neg x\}$. Pour simplifier l'écriture, les messages sont notés sans parenthèses : !!($x = \texttt{FALSE}$) sera noté !!$x = \texttt{FALSE}$. On remarque que l'automate est étiqueté par les propositions indiquant la

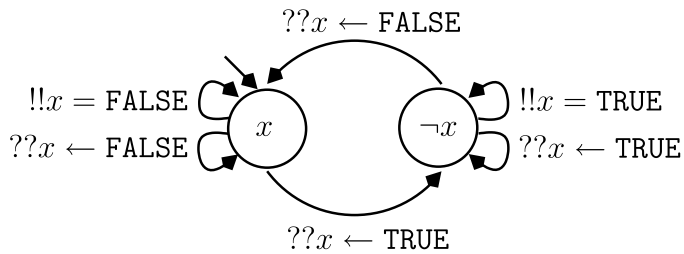

FIG. 4.24.: Automate A_x représentant une variable.

valeur de la variable. Les transitions de l'automate A_x sont de type broadcast, de telle sorte qu'elles sont conservées après synchronisation.

Si v est un symbole de variable, l'automate $A^V[v]$ représentant la variable v est

$$A^V[v] \stackrel{def}{=} A_x\{x \rightarrow v\}\{x \Rightarrow v\}$$

4.3.3.2. Expressions

Nous appelons automates d'expression les automates relevant d'une des deux formes suivantes :

- les automates des éléments de langage pour les expressions. Ces automates sont des constantes et sont utilisés, à l'aide de la synchronisation et du renommage, pour former l'automate représentant une expression. Ils sont au nombre de 6, un par forme d'expression du langage ST_0 (voir section 2.4.4 page 37) :
  - A^E_{var} est l'automate d'une variable vue comme une expression,
  - $A^E_{\mathtt{TRUE}}$ est l'automate de l'expression vraie,
  - $A^E_{\mathtt{FALSE}}$ est l'automate de l'expression fausse,
  - $A^E_{\neg}$ est l'automate de la négation,
  - $A^E_{\wedge}$ est l'automate de la conjonction,
  - $A^E_{\vee}$ est l'automate de la disjonction ;
- les automates représentant les expressions. À chaque expression e correspond un automate $A[e]$, formé inductivement à partir des automates d'éléments de langage. Par exemple, à l'expression $a \vee (b \wedge \mathtt{TRUE})$ correspond un automate, formé par la synchronisation entre l'automate représentant l'expression $(b \wedge \mathtt{TRUE})$, l'automate représentant l'expression a (formé à partir de l'automate de variable) et l'automate de la disjonction. L'automate représentant l'expression $(b \wedge \mathtt{TRUE})$ est lui-même formé par la synchronisation entre l'automate représentant l'expression b, l'automate de l'expression vraie et l'automate de la conjonction.

La figure 4.25 indique l'interface des automates d'expression, sans prendre en compte les messages concernant les variables. Les automates d'expression peuvent en effet contenir également des messages de type $?x = \mathtt{FALSE}$ ou $?x = \mathtt{TRUE}$ où x est une variable (il s'agit de messages de lecture de valeurs de variables ; il n'y a pas de message d'affectation de variable dans les expressions ST). De manière générale, un automate d'expression quitte son état

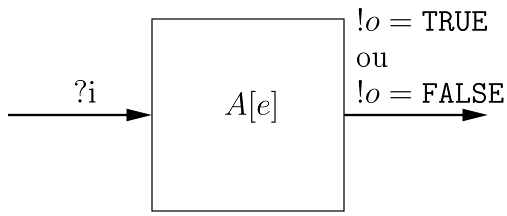

FIG. 4.25.: Interface (sans les messages liés aux variables) des automates représentant les expressions.

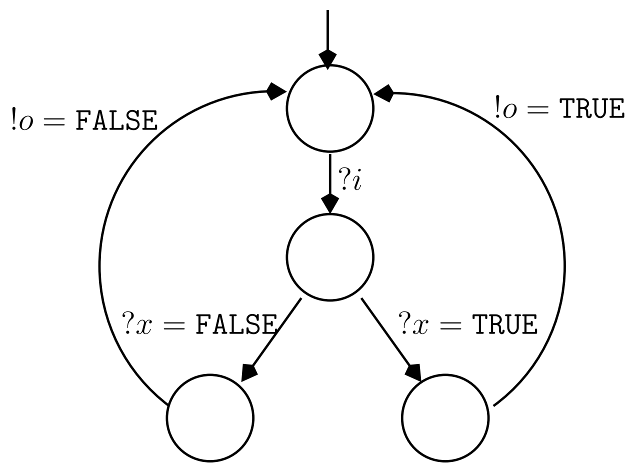

FIG. 4.26.: Automate A^E_{var} d'une variable vue comme une expression.

initial en se synchronisant sur un message $!i$, puis, après avoir franchi plusieurs transitions, le retrouve en se synchronisant sur un des deux messages $?o =$ FALSE ou $?o =$ TRUE. Cette interface représente un protocole d'appel et de réponse, la réponse portant une valeur booléenne.

À toute expression e, on associe un automate, que l'on note $A^E[e]$. $A^E[e]$ est défini inductivement sur la structure de e, par la synchronisation des éléments de langage qui la composent. Nous donnons ci-dessous la définition de $A^E[e]$ pour chaque type d'expression.

Variable vue comme une expression L'automate A^E_{var} d'une variable vue comme une expression est représenté à la figure 4.26. Il ne se synchronise pas directement avec les automates de variables car la synchronisation est effectuée globalement à un plus haut niveau. Le nom réel de la variable sera alors introduit au moyen d'un renommage.

L'automate A^E_{var} lit la valeur de la variable et la transmet sur la sortie.

Constantes L'automate $A^E_{\texttt{TRUE}}$ est représenté figure 4.27 ; l'automate $A^E_{\texttt{FALSE}}$ est représenté figure 4.28.

Négation L'automate $A^E_{\neg}$ est représenté à la figure 4.29.

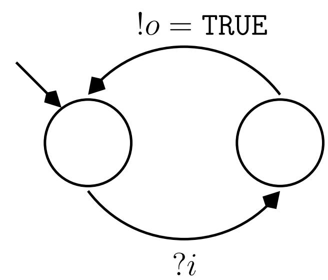

Fig. 4.27.: Automate $A^E_{\texttt{TRUE}}$ de l'expression vraie.

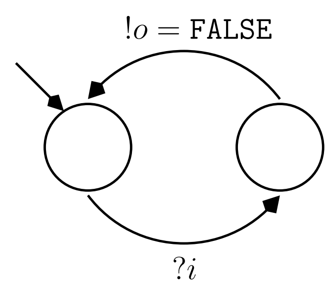

Fig. 4.28.: Automate $A^E_{\texttt{FALSE}}$ de l'expression fausse.

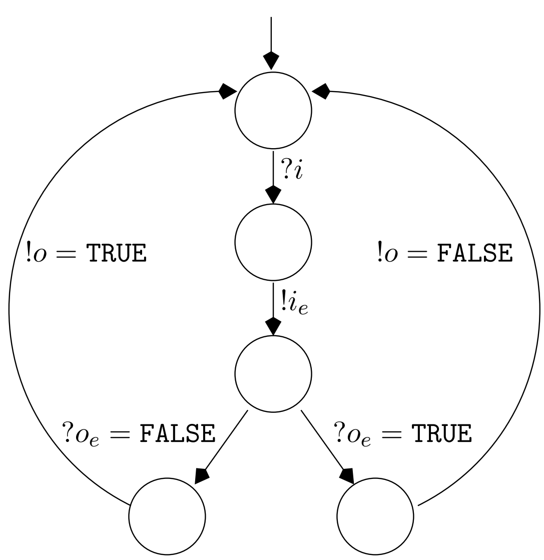

Fig. 4.29.: Automate $A^E_{\neg}$ de la négation.

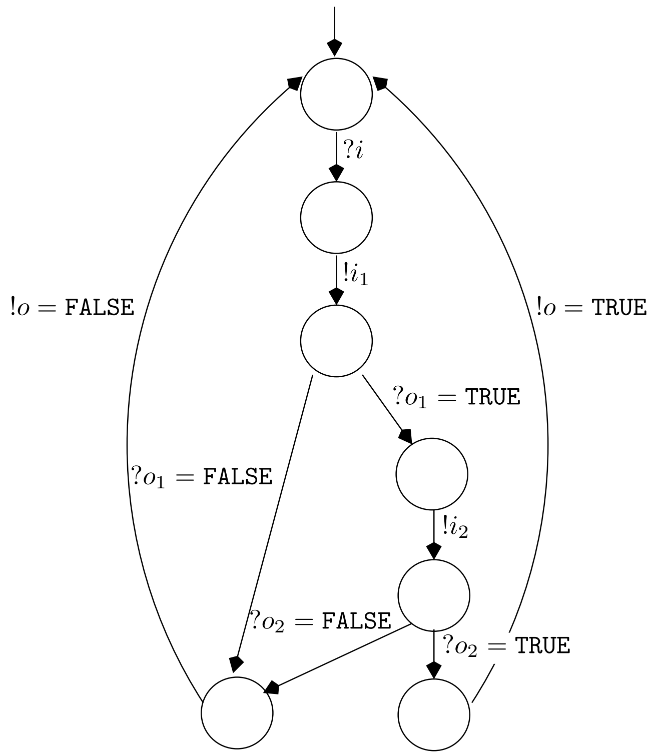

FIG. 4.30.: Automate $A^E_\wedge$ de la conjonction.

L'automate $A^E_\neg$ envoie un message qui déclenche l'évaluation de l'expression e, puis renvoie la négation du résultat. Pour toute expression e, l'automate $A[\neg e]$ est défini par

$$A^E[\neg e] \stackrel{def}{=} A^E_\neg \otimes A^E[e]\{i \to i_e, o \to o_e\}$$

Conjonction L'automate $A^E_\wedge$ est représenté à la figure 4.30.

L'automate $A^E_\wedge$ évalue la première expression et n'évalue la deuxième que si la première est vraie. Pour toutes expressions e_1 et e_2, l'automate $A^E[e_1 \wedge e_2]$ est défini par

$$\begin{aligned} A^E[e_1 \wedge e_2] \stackrel{def}{=} \; & A^E_\wedge \\ & \otimes(A^E[e_1]\{i \to i_1, o \to o_1\} \\ & \quad \otimes A^E[e_2]\{i \to i_2, o \to o_2\}) \end{aligned}$$

Par construction, les automates $A^E[e_1]$ et $A^E[e_2]$, après renommage, sont indépendants. Nous leur donnons donc la priorité pour la synchronisation (sur la synchronisation d'automates indépendants, voir la section 4.3.2.8 page 70). Le fait de les synchroniser ne supprimera donc aucun message, et la synchronisation de leur produit avec $A^E_\wedge$ ne posera aucun problème.

Disjonction L'automate $A^E_\vee$ est représenté à la figure 4.31.

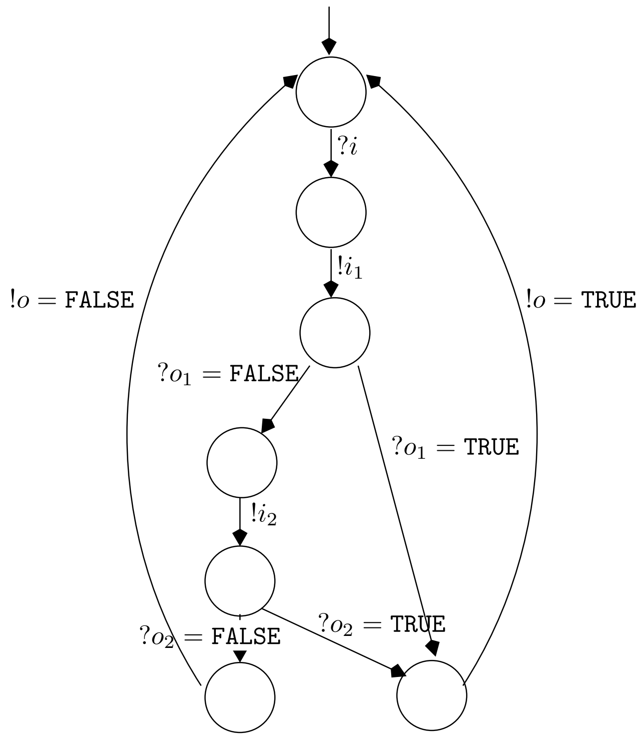

FIG. 4.31.: Automate $A^E_\vee$ de la disjonction

L'automate $A^E_\vee$ évalue la première expression et n'évalue la deuxième que si la première est fausse. Pour toutes expressions e_1 et e_2, l'automate $A[e_1 \vee e_2]$ est défini par

$$\begin{array}{rcl} A^E[e_1 \vee e_2] & \stackrel{def}{=} & A^E_\vee \\ & & \otimes(A^E[e_1]\{i \to i_1, o \to o_1\} \\ & & \quad \otimes A^E[e_2]\{i \to i_2, o \to o_2\}) \end{array}$$

L'ordre donné aux synchronisations suit la même règle que pour la conjonction.

4.3.3.3. Énoncés

Nous appelons automates d'énoncé les automates relevant d'une des deux formes suivantes :

- les automates des éléments de langage pour les énoncés. Ces automates sont des constantes et sont utilisés, à l'aide de la synchronisation et du renommage, pour former l'automate représentant un énoncé. Ils sont au nombre de 7, un par forme d'expression du langage ST_0 (voir section 2.4.4 page 37) :
  - $A^S_{\mathbf{skip}}$ est l'automate de l'énoncé vide,
  - $A^S_{\mathbf{exit}}$ est l'automate de l'énoncé `EXIT`,
  - $A^S_{\mathbf{return}}$ est l'automate de l'énoncé `RETURN`,
  - $A^S_{\leftarrow}$ est l'automate de l'énoncé d'affectation,

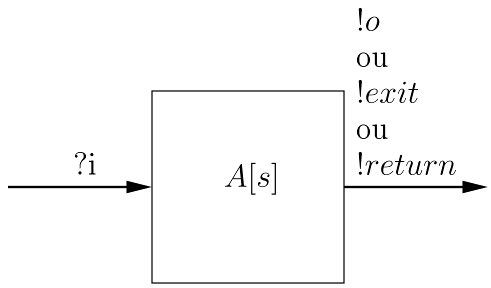

FIG. 4.32.: Interface (sans les messages liés aux variables) des automates représentant les énoncés.

  - $A^S_{\triangleright}$ est l'automate de l'énoncé de séquence,
  - $A^S_{\mathbf{if}}$ est l'automate de l'énoncé de sélection,
  - $A^S_{\mathbf{while}}$ est l'automate de l'énoncé d'itération ;
- les automates représentant les énoncés. À chaque énoncé s correspond un automate $A[s]$, formé inductivement à partir des automates d'éléments de langage.

La figure 4.32 indique l'interface des automates d'énoncé, sans prendre en compte les messages concernant les variables. Les automates d'énoncé peuvent en effet contenir également des messages de type $!x \leftarrow \texttt{FALSE}$, $!x \leftarrow \texttt{TRUE}$, $?x = \texttt{FALSE}$ ou $?x = \texttt{TRUE}$ où x est une variable. De manière générale, un automate d'énoncé quitte son état initial en se synchronisant sur un message $!i$, puis, après avoir franchi plusieurs transitions, doit finir[8] par retrouver son état initial en se synchronisant sur un message $?o$, $?exit$ ou $?return$. Cette interface représente un protocole d'appel et de réponse, la réponse informant sur la nature de la terminaison : un automate retrouvant son état d'attente sur le message $!o$ indique une terminaison normale. Un message $!exit$ indique qu'un énoncé **exit** a été rencontré (et non consommé[9]) dans un énoncé appelé, un message $!return$ indique qu'un énoncé **return** a été rencontré dans un énoncé appelé.

À tout énoncé s, on associe un automate, que l'on note $A^S[s]$. $A^S[s]$ est défini inductivement sur la structure de s, par la synchronisation des éléments de langage qui le composent. Nous donnons ci-dessous la définition de $A^S[s]$ pour chaque type d'énoncé.

Énoncé vide L'automate $A^S_{\mathbf{skip}}$ de l'énoncé vide est représenté à la figure 4.33. Il renvoie systématiquement le signal de terminaison normale.

Énoncé exit L'automate $A^S_{\mathbf{exit}}$ de l'énoncé **exit** est représenté à la figure 4.34. Il renvoie systématiquement le signal de terminaison par **exit**.

Énoncé return L'automate $A^S_{\mathbf{return}}$ de l'énoncé **return** est représenté à la figure 4.35. Il

[8]Dans certains cas (boucles), cela peut ne pas terminer. Ce genre de situations peut typiquement être détecté lors de la vérification de propriétés.

[9]On dit qu'un énoncé **exit** a été consommé lorsqu'il a été utilisé dans un énoncé **while**, voir la sémantique de l'énoncé d'itération page 86.

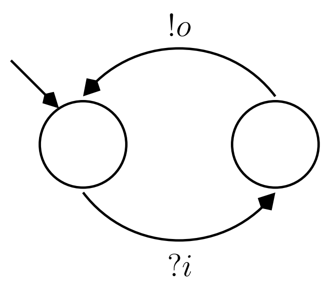

Fig. 4.33.: Automate $A^S_{\mathbf{skip}}$ de l'énoncé vide.

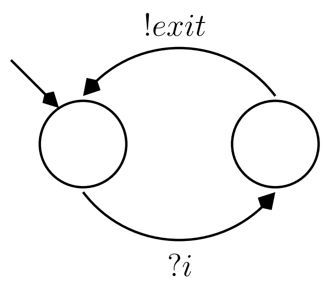

Fig. 4.34.: Automate $A^S_{\mathbf{exit}}$ de l'énoncé **exit**.

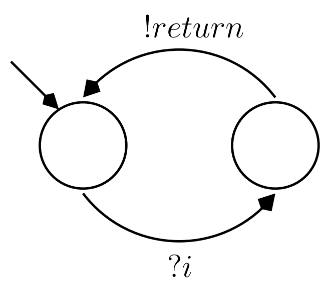

Fig. 4.35.: Automate $A^S_{\mathbf{return}}$ de l'énoncé **return**.

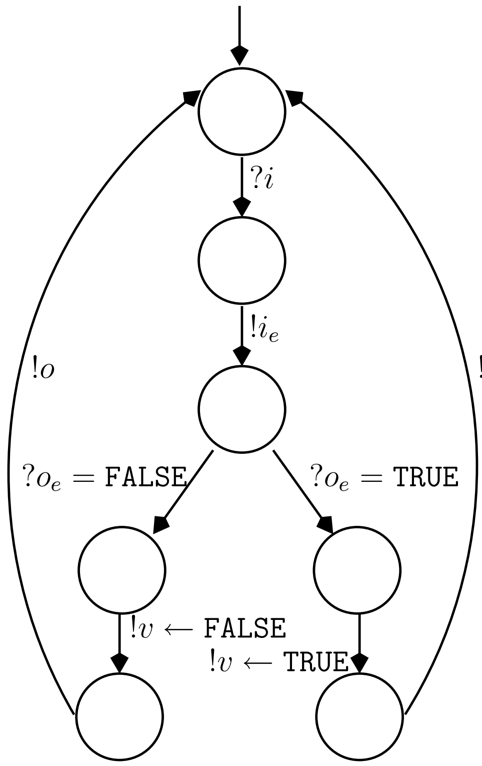

FIG. 4.36.: Automate $A^S_\leftarrow$ de l'affectation.

renvoie systématiquement le signal de terminaison par **return**.

Affectation L'automate $A^S_\leftarrow$ est représenté à la figure 4.36. Il effectue un appel correspondant à l'évaluation de l'expression e et envoie à la variable un message d'affectation du résultat. Il renvoie toujours un signal de terminaison normale.

Pour toute expression e, si x est le symbole de la variable à affecter[10], l'automate $A^S[x \leftarrow e]$ est défini par

$$A^S[x \leftarrow e] \stackrel{def}{=} A^S_\leftarrow\{v \rightarrow x\} \otimes A^E[e]\{i \rightarrow i_e, o \rightarrow o_e\}$$

Séquence L'automate $A^S_\triangleright$ est représenté à la figure 4.37. Il déclenche par le message $!i_i$ l'exécution de l'énoncé s_1. Si s_1 se termine sur un signal indiquant une terminaison par **exit** ou **return**, l'automate $A^S_\triangleright$ transmet ce signal et revient à son état d'attente, sans déclencher l'exécution de s_2. Si s_1 s'est terminé par un signal de terminaison normale, l'énoncé s_2 est exécuté et son signal de terminaison directement transmis. Le fait que l'énoncé de sélection ne déclenche pas l'exécution du deuxième énoncé en cas de terminaison du premier par **exit** ou **return** est fondamental. En effet, les deux énoncés **exit** et **return** se ont pour

[10]On rappelle que la synchronisation avec les variables ne se fait pas à ce niveau, voir la section 4.3.3.1 page 75.

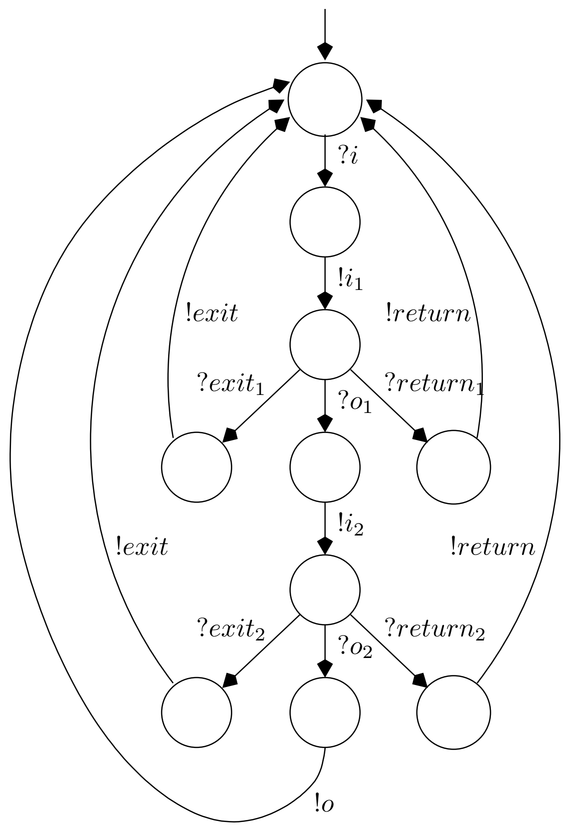

FIG. 4.37.: Automate $A^S_{\triangleright}$ représentant la séquence d'énoncés.

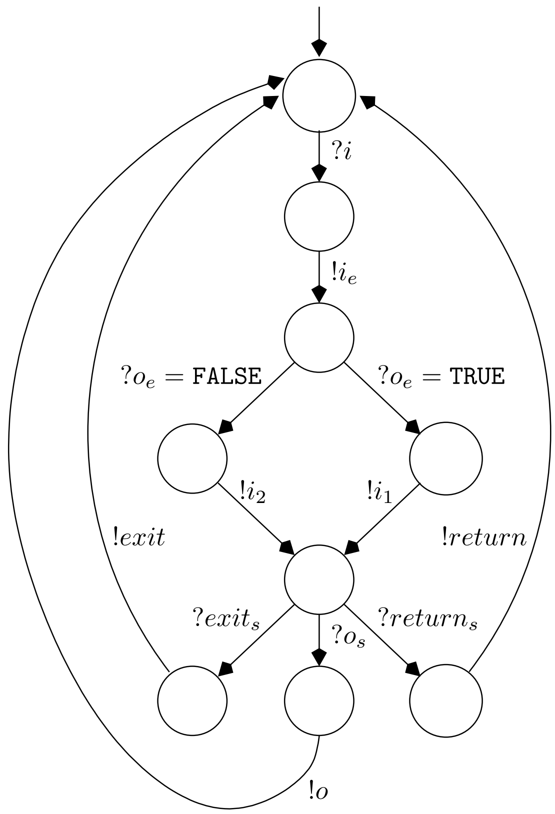

FIG. 4.38.: Automate $A^S_{\textbf{if}}$ représentant l'énoncé de sélection.

effet de stopper l'exécution de l'énoncé. Pour tous énoncés s_1 et s_2, l'automate $A^S[s_1 \triangleright s_2]$ est défini par

$$
\begin{aligned}
A^S[s_1 \triangleright s_2] \quad \stackrel{def}{=} \quad & A^S_{\triangleright} \\
& \otimes(A^S[s_1]\{i \rightarrow i_1, o \rightarrow o_1, exit \rightarrow exit_1, return \rightarrow return_1\} \\
& \quad \otimes A^S[s_2]\{i \rightarrow i_2, o \rightarrow o_2, exit \rightarrow exit_2, return \rightarrow return_2\})
\end{aligned}
$$

De même que pour la conjonction de deux expressions, la priorité est donnée à la synchronisation entre $A^S[s_1]$ et $A^S[s_2]$, car, par construction, ces automates sont indépendants après renommage.

Sélection L'automate $A^S_{\textbf{if}}$ est représenté à la figure 4.38. Il déclenche par le message $!i_e$ l'évaluation de l'expression e. Le résultat de l'évaluation de e conditionne le message qui déclenche l'exécution d'un des deux énoncés s_1 et s_2. Quel que soit l'énoncé exécuté, le signal de terminaison est directement transmis. Pour toute expression e, tous énoncés s_1

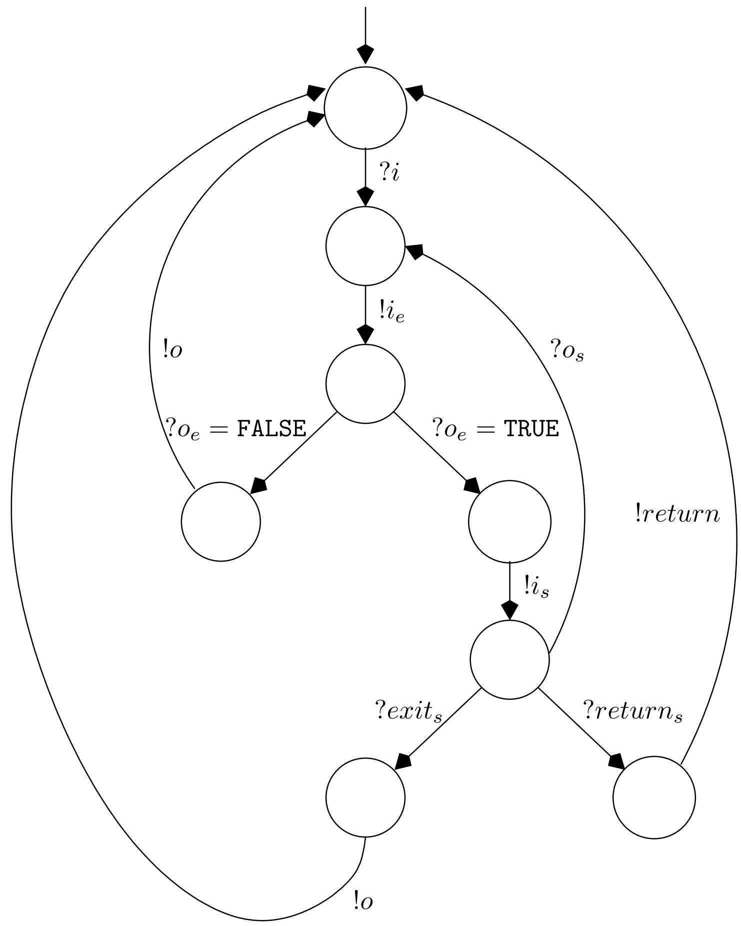

FIG. 4.39.: Automate $A^S_{\mathbf{while}}$ représentant l'énoncé d'itération.

et s_2, l'automate $A^S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]$ est défini par

$$
\begin{aligned}
A^S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \stackrel{def}{=}\ & A^S_{\mathbf{if}} \\
& \otimes(A^E[e]\{i \to i_e, o \to o_e\} \\
& \quad \otimes A^S[s_1]\{i \to i_1, o \to o_s, exit \to exit_s, return \to return_s\} \\
& \quad \otimes A^S[s_2]\{i \to i_2, o \to o_s, exit \to exit_s, return \to return_s\})
\end{aligned}
$$

La priorité est donnée à la synchronisation entre $A^E[e]$, $A^S[s_1]$ et $A^S[s_2]$, car après renommage ces automates sont indépendants (pour cette raison, il n'est pas nécessaire d'indiquer une priorité pour la synchronisation entre ces trois automates).

Itération L'automate $A^S_{\mathbf{while}}$ est représenté à la figure 4.39. Il effectue en boucle l'évaluation de l'énoncé e et, selon le résultat, l'exécution de l'énoncé s. Si l'énoncé s se termine avec le signal de terminaison **return**, la boucle est stoppée et le signal transmis. Si l'énoncé s se termine avec le signal **exit**, la boucle est stoppée mais c'est un signal de terminaison

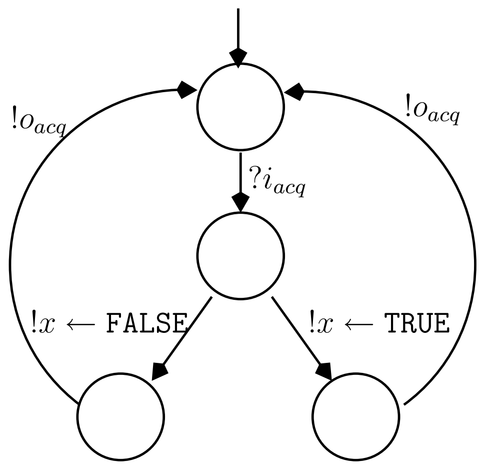

FIG. 4.40.: Automate A_{ent} représentant l'automate d'une variable d'entrée.

normale qui est transmis (le signal **exit** a été consommé par le **while** le plus profond, c'est-à-dire celui vers lequel il est directement « remonté »). Pour toute expression e et tout énoncé s, l'automate $A^S[\textbf{while}\ e\ \textbf{do}\ s]$ est défini par

$$\begin{aligned} A^S[\textbf{while}\ e\ \textbf{do}\ s] \stackrel{def}{=}\ & A^S_{\textbf{while}} \\ & \otimes(A^E[e]\{i \to i_e, o \to o_e\} \\ & \quad \otimes A^S[s]\{i \to i_s, o \to o_s, exit \to exit_s, return \to return_s\}) \end{aligned}$$

La priorité est donnée à la synchronisation entre $A^E[e]$ et $A^S[s]$, car après renommage ces automates sont indépendants.

4.3.3.4. Interprétation d'un programme

Acquisition des variables L'automate A_{ent} représentant l'acquisition d'une variable d'entrée est représenté figure 4.40.

Si v est un symbole de variable d'entrée, l'automate $A^{acq}[v]$ représentant l'acquisition de v est donné par

$$A^{acq}[v] \stackrel{def}{=} A_{ent}\{x \leftarrow v\}$$

Interpréteur du programme L'automate A_p de l'interpréteur de programmes est représenté figure 4.41.

Si p est un programme, c'est aussi un énoncé s représenté par un automate $A^S[s]$. Soit Σ_p l'ensemble $\{v_1, \ldots, v_n\}$ des variables déclarées du système ; on distingue les variables d'entrées : il existe un i tel que, pour tout $j \leq i$, $v_j \in \Sigma_{in}$. L'automate $A^P[p]$ effectuant

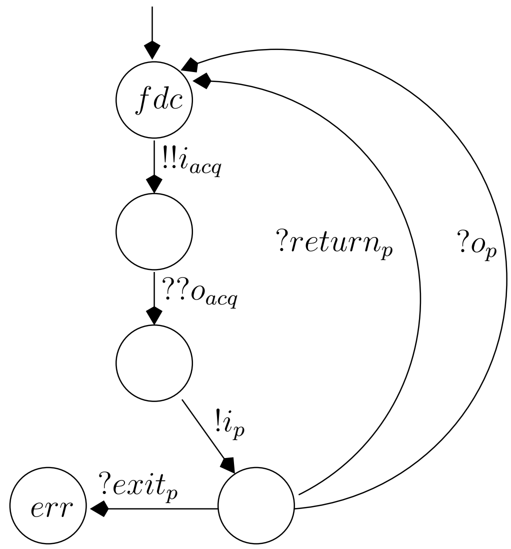

FIG. 4.41.: Automate interpréteur d'un programme ST_0.

l'interprétation du programme p est donné par

$$\begin{aligned} A^P[p] \quad \stackrel{def}{=} \quad & A^S[s]\{i \rightarrow i_p, o \rightarrow o_p, exit \rightarrow exit_p, return \rightarrow return_p\} \\ & \otimes((A^{acq}[v_1] \otimes \ldots \otimes A^{acq}[v_i]) \\ & \quad \otimes(A^V[v_1] \otimes \ldots \otimes A^V[v_n])) \end{aligned}$$

4.3.4. Conclusion

Le formalisme qui précède a permis de définir une sémantique modulaire prenant en compte les programmes écrits simultanément dans trois des langages de la norme, dont ST. Cette sémantique offre l'avantage d'être plus claire que la sémantique directe ST. La définition de la structure des programmes est naturelle, y compris la prise en compte de la capture de continuation (énoncés **exit** et **return**).

Cette sémantique n'est cependant pas aussi compacte que la précédente : le système de transitions associé à un programme donné est de taille bien plus grande. Ce système se code assez directement (voir la section 6.4 page 124) dans un model-checker, mais ce codage exigera probablement d'être optimisé.

4.4. Prise en compte des entiers bornés

Les modèles proposés donnent une sémantique opérationnelle pour les langages IL_0 et ST_0, qui ne prennent en compte que le type booléen. La prise en compte des entiers bornés peut être effectuée de différentes façons :

1. par transformation des entiers en booléens à la compilation. Cette méthode est simple

à réaliser et elle est applicable à tous les modèles ; elle a le défaut de rendre moins traçable le code obtenu à partir du modèle formel ;

2. par extension de la sémantique aux entiers bornés.

Cette dernière n'est pas toujours utile à réaliser. Par exemple, la structure du langage ST fait que la définition d'une sémantique prenant en compte les entiers bornés représente une augmentation importante des définitions et des outils formels, qui ne font qu'alourdir le modèle existant (un luxe que ne peut se payer la sémantique directe ST_0). Il est possible, dans le cas de la sémantique modulaire, de représenter des entiers en synchronisant des automates booléens entre eux. Cela revient peu ou prou à appliquer la première solution, qui a l'avantage de laisser sa clarté au modèle d'origine.

La simplicité du langage IL, cependant, permet d'envisager une extension aux entiers bornés de sa sémantique directe, sans pour autant la modifier fondamentalement. Cette sémantique a été présentée brièvement dans [CDP+00] et [CCL+00] ; nous la décrivons entièrement à l'annexe B page 179. Moyennant l'ajout d'un état d'erreur (pour les dépassements et les erreurs de typage), elle permet, tout en apportant une extension utile, de préserver la simplicité et la compacité du modèle d'origine.

4.5. Conclusion

Nous avons proposé deux formalismes différents pour exprimer la sémantique opérationnelle des langages textuels :

– une sémantique à base d'automates non étiquetés, où le système de transitions associé à un programme est déduit en bloc du programme entier ;
– une sémantique modulaire à base d'automates étiquetés synchronisés par envoi de messages, où le système de transitions est déduit de manière inductive, par construction sur les éléments de langage qui forment le programme de départ.

La première sémantique est particulièrement adaptée au cas du langage IL et permet de définir un modèle aussi simple que compact. Le cas de ST est plus délicat, et nous avons vu que le compromis entre compacité, simplicité et prise en compte du multilangage est affaire de choix.

Il est à noter que les deux modèles ont en commun une granularité très fine : les transitions du système représentent l'exécution d'une instruction, voire même une partie d'une instruction. Cela permet une visibilité très détaillée du comportement d'un programme, mais pourra rendre plus difficile la vérification de propriétés sur le comportement observable de l'extérieur, c'est-à-dire des états du système à la fin de chaque cycle (avant les acquisitions d'entrée). Dans la suite, nous discuterons de la vérification du comportement observable d'un programme d'API.

5. Vérification des modèles de programmes avec SMV

5.1. Introduction

Étant donné un modèle de programme, obtenu à partir de la sémantique opérationnelle du langage dans lequel il est écrit, nous utilisons la technique du model-checking symbolique pour vérifier des propriétés sur son comportement. Nous utilisons pour cela le logiciel SMV, qui effectue la vérification d'un système décrit sous la forme d'un système de transitions, au moyen de propriétés écrites en logique temporelle. Nous décrivons dans ce chapitre les principes et les techniques mis en œuvre dans le processus de vérification ; après avoir introduit le model-checking dans son principe, nous décrivons les logiques temporelles et comment les utiliser pour écrire des propriétés ; puis nous présentons les principes et la mise en œuvre de la spécification d'un système avec le logiciel SMV.

Logique temporelle Les logiques temporelles permettent d'exprimer des propriétés dynamiques portant sur l'exécution du système que l'on souhaite vérifier. Nous utilisons, dans cette étude, les deux logiques LTL et CTL, décrites respectivement aux sections 5.2.2 page 93 et 5.2.3 page 95.

Le choix entre les deux logiques est dépendant, entre autres, de l'outil utilisé, du champ d'application, et des propriétés que l'on souhaite exprimer.

Model-checking symbolique Étant donné le modèle A d'un système (programme, système automatique, protocole...) et une propriété φ, le problème du model-checking (décrit notamment dans [CGP99, SBB$^+$99]) revient à répondre à la question : « le modèle A vérifie-t'il la propriété φ ? ».

Le modèle est formel, c'est-à-dire qu'il est spécifié mathématiquement, et représente le comportement du système réel qu'il modélise (ou une partie de ce comportement, de manière plus ou moins abstraite). De même, les propriétés sont spécifiées formellement.

L'efficacité des algorithmes est primordiale ; en effet, la résolution d'un problème de model-checking suppose une exploration exhaustive des états accessibles du système étudié. Si on considère, comme c'est par exemple le cas de la sémantique décrite à la section 4.2.3 page 49, que les états sont constitués d'une valuation des variables, il apparaît clairement que le nombre d'états du système est exponentiel par rapport au nombre de variables ; cela rend hors de portée la vérification exhaustive de ces états.

La technique du model-checking symbolique (mise en œvre par SMV), par l'utilisation de méthodes puissantes de représentation, permet d'effectuer la vérification de systèmes de très grande taille [JEK+92, CGL92]. Cela est également dû à l'existence d'algorithmes performants pour le model-checking de propriétés écrites en logique temporelle, y compris pour les systèmes représentés symboliquement.

5.2. Logiques temporelles

5.2.1. Introduction

Les logiques temporelles (voir [SBB+99, McM93]) permettent d'exprimer des propriétés dynamiques portant sur l'exécution du système que l'on souhaite vérifier. Ces propriétés sont exprimées par des formules utilisant des propositions atomiques, définies en fonction du système. Dans notre cas, ces propositions atomiques sont les suivantes :

- pour la sémantique directe, les propositions sont en rapport avec les valeurs des variables pour chaque état, et la valeur de l'accumulateur (pour $x =$ `TRUE` on écrira x et pour $x =$ `FALSE` on écrira $\neg x$) ;
- pour la sémantique modulaire, ce sont les étiquettes placées sur les états (ce sont les valeurs des variables[1] et les étiquettes fdc et err de l'interpréteur principal[2]).

Les logiques temporelles fournissent en général plusieurs opérateurs, qui peuvent être imbriqués d'une manière arbitraire. Parmi ces opérateurs figurent les opérateurs booléens classiques : $\vee$, $\wedge$, $\neg$. De plus, les logiques temporelles proposent des « combinateurs temporels », qui permettent d'exprimer des propriétés portant sur l'évolution des événements au cours de l'exécution du système. Il est donc, par exemple, possible d'exprimer des propriétés comme :

> *Le moteur n'est jamais actionné en même temps que le frein.*
>
> *Le frein se déclenche inévitablement après que la pédale a été enfoncée.*
>
> *L'alarme se déclenche toujours après un signal d'alerte, à moins qu'un bouton d'annulation n'ait été appuyé.*

Il existe de nombreuses logiques temporelles, dont les pouvoirs d'expression et les performances[3] varient. Nous présentons ici deux logiques apparentées, mais dont les pouvoirs d'expression diffèrent. Les deux sont couramment utilisées dans le domaine du model-checking ; nous verrons par la suite quel rôle chacune peut jouer dans la vérification des programmes d'automates industriels.

[1] Voir la section 4.3.3.1 page 75 pour la définition des automates de variables.
[2] L'automate de l'interpréteur principal est défini à la section 4.3.3.4 page 87.
[3] Par performances, nous entendons notamment la complexité du model-checking.

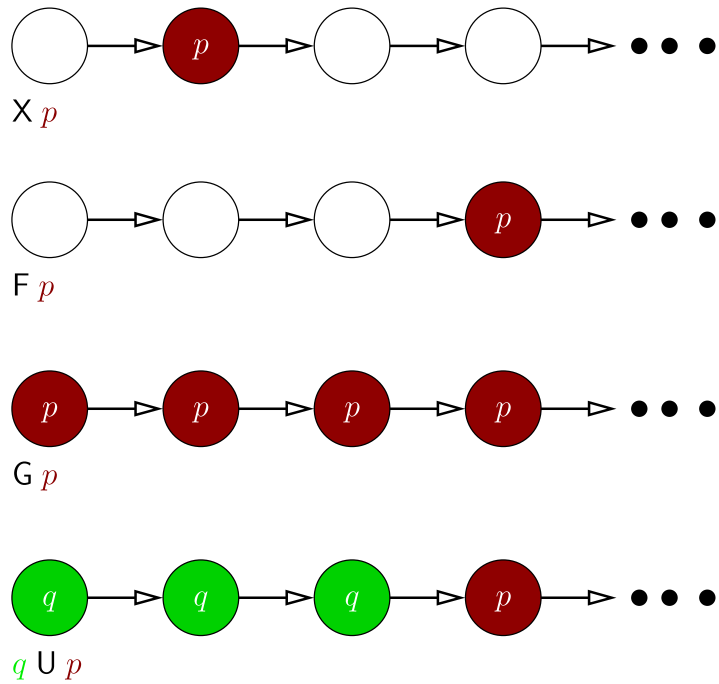

FIG. 5.1.: Les combinateurs de la logique LTL.

5.2.2. LTL

5.2.2.1. Présentation

La logique LTL (voir par exemple [Eme90, McM]) exprime des propriétés le long d'un chemin. On considère donc une seule exécution du système. C'est au cours de cette exécution que se produisent des événements sur lesquels nous pouvons exprimer des propriétés. La logique LTL met à notre disposition les combinateurs temporels X, G, F et U, dont la figure 5.1 donne une idée intuitive, qui peut se résumer comme suit :

- la formule $\mathsf{X}\ \varphi$ (*à l'instant suivant,* φ) est vraie à l'instant t si φ est vraie à l'instant $t+1$;
- la formule $\mathsf{G}\ \varphi$ (*toujours* φ) est vraie à l'instant t si φ est vraie pour tout instant $t' \geq t$;
- la formule $\mathsf{F}\ \varphi$ (*inévitablement,* φ) est vraie à l'instant t si φ est vraie à un instant $t' \geq t$;
- la formule $\varphi_1\ \mathsf{U}\ \varphi_2$ (φ_1 *jusqu'à* φ_2) est vraie à un instant t si la formule φ_2 est vraie à un instant $t' \geq t$ **et** si φ_1 est vraie à tout instant entre t et t'.

Comme on le constate, la notion de temps est discrète et correspond en fait aux transitions du système. Par « à l'instant suivant », il faut donc comprendre « à l'état suivant ».

5.2.2.2. Définition formelle

Soit A un système de transitions dont Q est l'ensemble des états, $\rightarrow$ la relation de transition, Π un ensemble de propositions atomiques et l la fonction d'étiquetage des états (à tout état, l associe un sous-ensemble de Π[4]).

Soit x une exécution de A, c'est-à-dire une suite $q_0, q_1, \ldots$, telle que, pour tout i, $q_i \in Q$ et $q_i \rightarrow q_{i+1}$. Pour une exécution $x = q_0, q_1, \ldots$, on note x^k l'exécution *suffixe* de x, tel que $x^k = q_k, q_{k+1}, \ldots$. L'expression $x \models \varphi$ signifie « l'exécution x vérifie la formule φ ». On définit $\models$ inductivement sur la structure de la formule :

- $x \models \texttt{TRUE}$ et $x \not\models \texttt{FALSE}$;
- $x \models p$ (où p est une proposition atomique) ssi $p \in l(q_0)$;
- $x \models \varphi_1 \vee \varphi_2$ ssi $x \models \varphi_1$ ou $x \models \varphi_2$;
- $x \models \varphi_1 \wedge \varphi_2$ ssi $x \models \varphi_1$ et $x \models \varphi_2$;
- $x \models \neg\varphi$ ssi $x \not\models \varphi$;
- $x \models \mathsf{X}\, \varphi$ ssi $x^1 \models \varphi$;
- $x \models \varphi_1 \,\mathsf{U}\, \varphi_2$ ssi $\exists j (x^j \models \varphi_2$ et $\forall k < j (x^k \models \varphi_1))$;
- $x \models \mathsf{F}\, \varphi$ ssi $\exists j (x^j \models \varphi)$;
- $x \models \mathsf{G}\, \varphi$ ssi $\forall j (x^j \models \varphi)$.

Afin de rendre les opérateurs explicites, nous en donnons une définition complète, mais, en pratique, il suffit de définir quelques-uns des opérateurs. Nous pouvons en particulier noter l'existence des relations $\mathsf{G}\, \varphi \equiv \neg \mathsf{F}\, \neg\varphi$ et $\mathsf{F}\, \varphi \equiv \texttt{TRUE}\, \mathsf{U}\, \varphi$. Pour définir complètement la logique LTL, il suffit par exemple de définir les opérateurs $\neg$, $\wedge$, X et U.

Il existe de plus des opérateurs supplémentaires, comme par exemple l'opérateur W (ou *weak until*), défini par $\varphi_1 \;\mathsf{W}\; \varphi_2 \equiv \mathsf{G}\, \neg\varphi_2 \vee (\varphi_1 \;\mathsf{U}\; \varphi_2)$. Il s'agit d'une variante de l'opérateur U (« jusqu'à ce que... » dans lequel la propriété φ_2 peut ne jamais être vraie. Nous utiliserons l'opérateur W, par la suite, pour vérifier des propriétés.

Exemple La propriété $pédale \Rightarrow \mathsf{F}\, frein$ signifie que, si la proposition *pédale* est vérifiée, la proposition $frein$ finit inévitablement par être vraie (on rappelle que l'opérateur $\Rightarrow$ représente l'implication logique, telle que $\varphi_1 \Rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$).

5.2.2.3. Définition de $A \models \varphi$

Soit A un système de transitions dont Q est l'ensemble des états, $I \subseteq Q$ l'ensemble des états initiaux, et soit φ une formule LTL. On dit que A vérifie la formule φ, noté $A \models \varphi$, si et seulement si

$$\text{pour toute exécution } x = q_0, q_1, \ldots \text{ avec } q_0 \in I,\ q \models \varphi$$

[4] On trouvera un exemple d'une telle définition à la section 4.3.2.5 page 67.

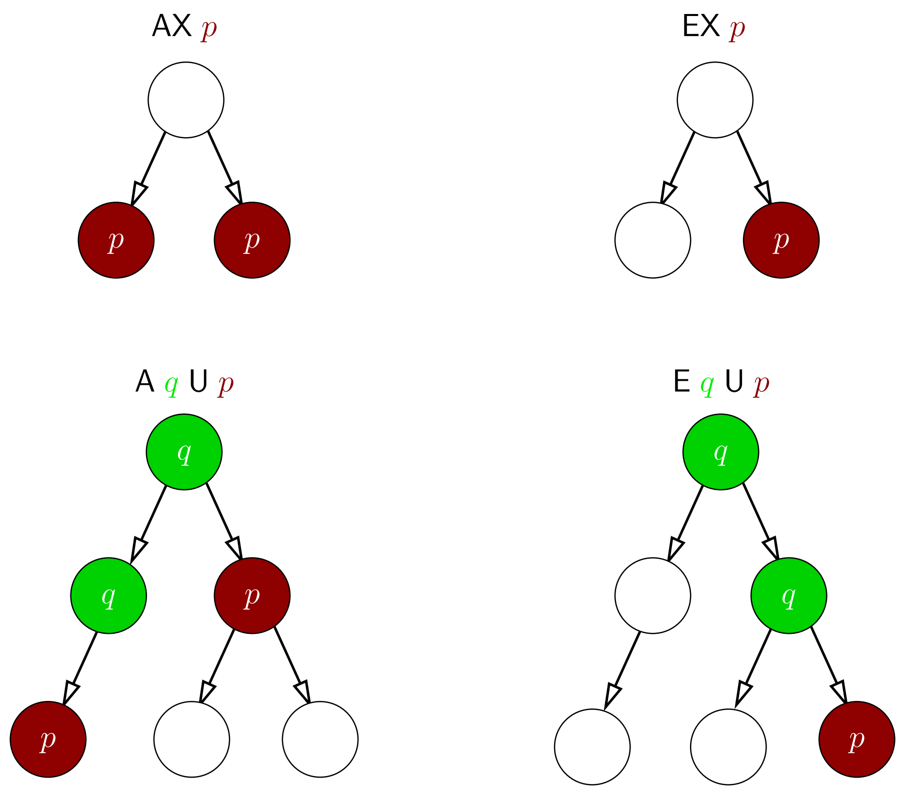

FIG. 5.2.: Les combinateurs AX, EX, A . . . U et E . . . U de la logique CTL.

5.2.3. CTL

5.2.3.1. Présentation

Alors que la logique LTL permet d'exprimer des événements survenant sur une exécution unique, la logique CTL permet, au contraire, d'envisager plusieurs possibilités (plusieurs futurs) et de les prendre en compte dans une même formule. Ainsi, on ne considère plus une seule exécution, mais une arborescence d'exécution (dont les nœuds correspondent aux bifurcations, c'est-à-dire aux futurs possibles). C'est pourquoi la logique CTL est qualifiée d'*arborescente*.

Une formule CTL ne s'applique donc pas à un chemin mais à un état. Les combinateurs temporels proposés intègrent la quantification sur les chemins, grâce à deux quantificateurs :

- le quantificateur A signifie : « tous les successeurs sont tels que... » ;
- le quantificateur E signifie : « il existe un successeur tel que... ».

La particularité de CTL est que tout quantificateur sur les chemins doit être utilisé en combinaison avec un combinateur d'états (ceux de LTL : X, F, G, etc.). Par conséquent, les opérateurs principaux de CTL, illustrés par la figure 5.2, peuvent être informellement décrits comme suit :

- la formule AX φ est vraie pour un état si φ est vraie pour tous ses successeurs ;
- la formule EX φ est vraie pour un état si φ est vraie pour l'un au moins de ses

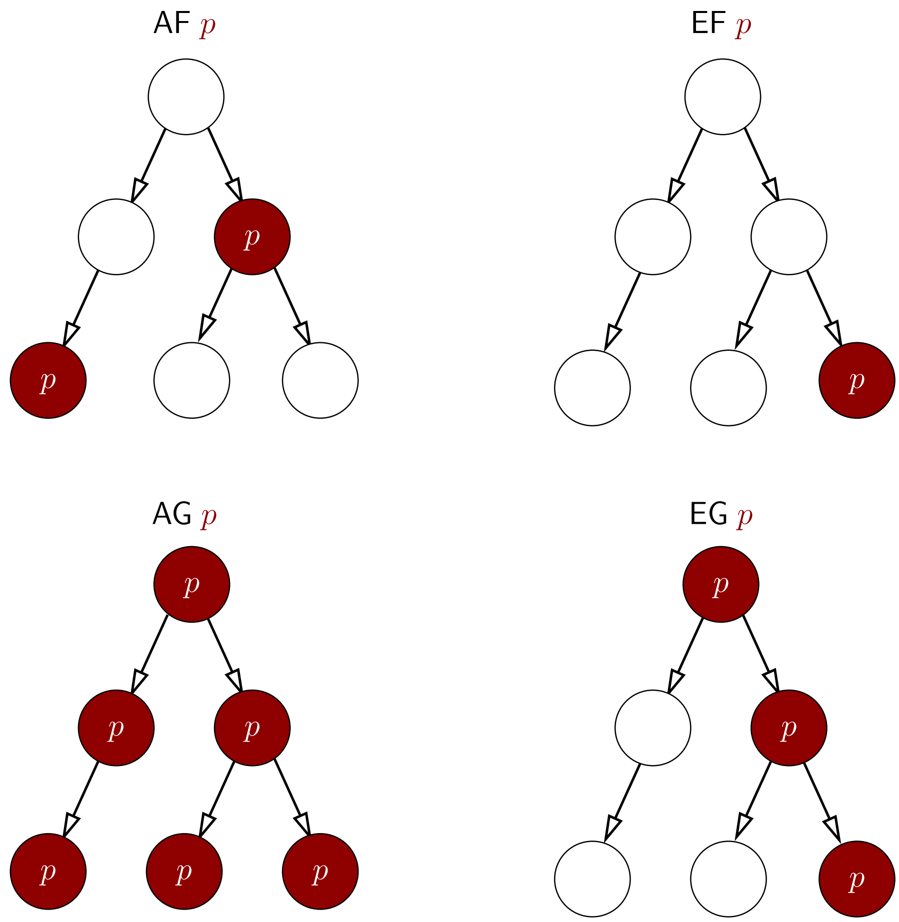

Fig. 5.3.: Les combinateurs AF, EF, AG et EG de la logique CTL.

successeurs ;
- la formule A φ_1 U φ_2 est vraie pour un état si tous les chemins partant de cet état vérifient φ_1 U φ_2 ;
- la formule E φ_1 U φ_2 est vraie pour un état s'il existe un chemin partant de cet état qui vérifie φ_1 U φ_2.

Il existe également AF, EF, AG et EG, que l'on peut définir à partir des autres combinateurs, grâce aux équivalences suivantes :

$$\begin{aligned}
\mathsf{AF}\ \varphi &\equiv \mathsf{A}\ \mathtt{TRUE}\ \mathsf{U}\ \varphi \\
\mathsf{EF}\ \varphi &\equiv \mathsf{E}\ \mathtt{TRUE}\ \mathsf{U}\ \varphi \\
\mathsf{AG}\ \varphi &\equiv \neg\mathsf{EF}\ \neg\varphi \\
\mathsf{EG}\ \varphi &\equiv \neg\mathsf{AF}\ \neg\varphi
\end{aligned}$$

Ainsi, la formule AF φ est vraie pour un état si tous les chemins à partir de cet état vérifient F φ, etc. La figure 5.3 illustre ces quatre combinateurs.

Les combinateurs sont relativement intuitifs, mais les relations entre eux sont complexes : il est, par exemple, bien connu que $\mathsf{A} \ldots \mathsf{U}$ peut être exprimé en fonction de AF et $\mathsf{E} \ldots \mathsf{U}$[5], mais [Lar95] a démontré que la réciproque (exprimer $\mathsf{E} \ldots \mathsf{U}$ en fonction de $\mathsf{A} \ldots \mathsf{U}$ et EF) était impossible.

5.2.3.2. Définition formelle

Soit A un système de transitions dont Q est l'ensemble des états, $\rightarrow$ la relation de transition, Π un ensemble de propositions atomiques et l la fonction d'étiquetage des états. Soit $q \in Q$ un état. L'expression $q \models \varphi$ signifie « l'état q vérifie la formule φ ». On définit $\models$ inductivement sur la structure de la formule :

- $q \models \texttt{TRUE}$ et $q \not\models \texttt{FALSE}$;
- $q \models p$ (où p est une proposition atomique) ssi $p \in l(q)$;
- $q \models \varphi_1 \vee \varphi_2$ ssi $q \models \varphi_1$ ou $q \models \varphi_2$;
- $q \models \varphi_1 \wedge \varphi_2$ ssi $q \models \varphi_1$ et $q \models \varphi_2$;
- $q \models \neg\varphi$ ssi $q \not\models \varphi$;
- $q \models \mathsf{AX}\ \varphi$ ssi pour tout q' tel que $q \rightarrow q'$, $q' \models \varphi$;
- $q \models \mathsf{EX}\ \varphi$ ss'il existe q' tel que $q \rightarrow q'$ et $q' \models \varphi$;
- $q \models \mathsf{A}\ \varphi_1\ \mathsf{U}\ \varphi_2$ ssi pour toute exécution $q_0, q_1, \ldots$ avec $q_0 = q$, il existe un i tel que :
  - $q_i \models \varphi_2$,
  - et pour tout j tel que $0 \leq j < i$, $q_j \models \varphi_1$;
- $q \models \mathsf{E}\ \varphi_1\ \mathsf{U}\ \varphi_2$ ss'il existe une exécution $q_0, q_1, \ldots$ avec $q_0 = q$, et un i tel que :
  - $q_i \models \varphi_2$,
  - et pour tout j tel que $0 \leq j < i$, $q_j \models \varphi_1$;
- $q \models \mathsf{AF}\ \varphi$ ssi pour toute exécution $q_0, q_1, \ldots$ avec $q_0 = q$, il existe un i tel que $q_i \models \varphi$;
- $q \models \mathsf{EF}\ \varphi$ ss'il existe une exécution $q_0, q_1, \ldots$ avec $q_0 = q$, et un i tel que $q_i \models \varphi$;
- $q \models \mathsf{AG}\ \varphi$ ssi pour toute exécution $q_0, q_1, \ldots$ avec $q_0 = q$, et pour tout i, on a $q_i \models \varphi$;
- $q \models \mathsf{EG}\ \varphi$ ss'il existe une exécution $q_0, q_1, \ldots$ avec $q_0 = q$, telle que pour tout i, on ait $q_i \models \varphi$.

5.2.3.3. Définition de $A \models \varphi$

Soit A un système de transitions dont Q est l'ensemble des états, $I \subseteq Q$ l'ensemble des états initiaux, et soit φ une formule CTL. On dit que A vérifie la formule φ, noté $A \models \varphi$, si et seulement si

$$\forall q \in I\ q \models \varphi$$

5.2.4. Comparaison des pouvoirs d'expression

Bien que proches et apparentées, LTL et CTL ont des pouvoirs d'expression bien distincts. Certes, il existe de nombreuses propriétés pouvant s'exprimer indifféremment dans l'une ou l'autre des logiques. Mais il existe également des propriétés qui peuvent s'exprimer

[5] La formule est $\mathsf{A}\ \varphi_1\ \mathsf{U}\ \varphi_2 \equiv \mathsf{AF}\ \varphi_2 \wedge \neg\mathsf{E}\ (\neg\varphi_2)\ \mathsf{U}\ (\neg\varphi_1 \wedge \neg\varphi_2)$.

dans l'une et pas dans l'autre. La comparaison entre LTL et CTL est une question délicate[6], qui a été abordée dans la littérature (notamment [EH86]). Nous nous contenterons d'illustrer, au travers de deux exemples, deux différences flagrantes, nous permettant de conclure qu'aucune des deux logiques ne peut se réduire à une partie de l'autre.

5.2.4.1. CTL $\not\subseteq$ LTL : atteignabilité

Une propriété d'atteignabilité exprime le fait qu'il existe une exécution du système vérifiant une propriété donnée. Par exemple :

À partir de l'état initial, il est possible d'atteindre la fin de cycle.

Le mot « possible » signifie ici qu'il existe une exécution dans laquelle la fin de cycle est atteinte. Cette propriété s'exprime $\mathsf{EF}\ fdc$ en CTL ; elle exploite la possibilité, donnée par l'opérateur E, de quantifier sur les chemins. Cela n'est clairement pas possible en LTL [EH86], où on ne peut qu'exprimer des propriétés le long d'une exécution donnée, et de cette seule exécution.

On peut en conclure que CTL ne peut pas se réduire à LTL, ou, autrement dit, que LTL n'est pas « plus grand » que CTL.

5.2.4.2. LTL $\not\subseteq$ CTL : hypothèses dynamiques

Nous prenons l'exemple de feux de signalisation disposés à un carrefour, représenté à la figure 5.4 page suivante. Ce carrefour est composé d'une route principale et d'une route secondaire. Par défaut, le feu de la route secondaire est rouge ; il se met au vert si une voiture vient à passer sur un capteur. Nous voulons exprimer en logique temporelle la propriété suivante :

Dans l'hypothèse où la route secondaire ne reste pas déserte infiniment longtemps, le feu principal passe régulièrement au rouge.

Il s'agit en fait d'une propriété de vivacité, permettant de vérifier que, sous réserve d'un fonctionnement normal des capteurs, les feux assurent la circulation chaque fois que nécessaire.

La phrase comporte deux propositions : la première peut s'énoncer en CTL par la formule AG AF `capteur_secondaire_activé`, et G F `capteur_secondaire_activé` en LTL. La seconde proposition s'écrit AG AF `feu_principal_rouge` en CTL et s'écrit en LTL G F `feu_principal_rouge`. La formule que l'on souhaite exprimer est une implication logique entre les deux. En LTL, cela s'écrit simplement

$$((\mathsf{G}\ \mathsf{F}\ \texttt{capteur\_secondaire\_activé}) \Rightarrow (\mathsf{G}\ \mathsf{F}\ \texttt{feu\_principal\_rouge}))$$

En revanche, cette propriété ne peut pas s'exprimer directement en CTL. Par exemple, la formule

$$((\mathsf{AG}\ \mathsf{AF}\ \texttt{capteur\_secondaire\_activé}) \Rightarrow (\mathsf{AG}\ \mathsf{AF}\ \texttt{feu\_principal\_rouge}))$$

[6] De par le fait qu'elles ne s'appliquent pas aux mêmes objets (chemins ou états), LTL et CTL se prêtent peu à une stricte comparaison

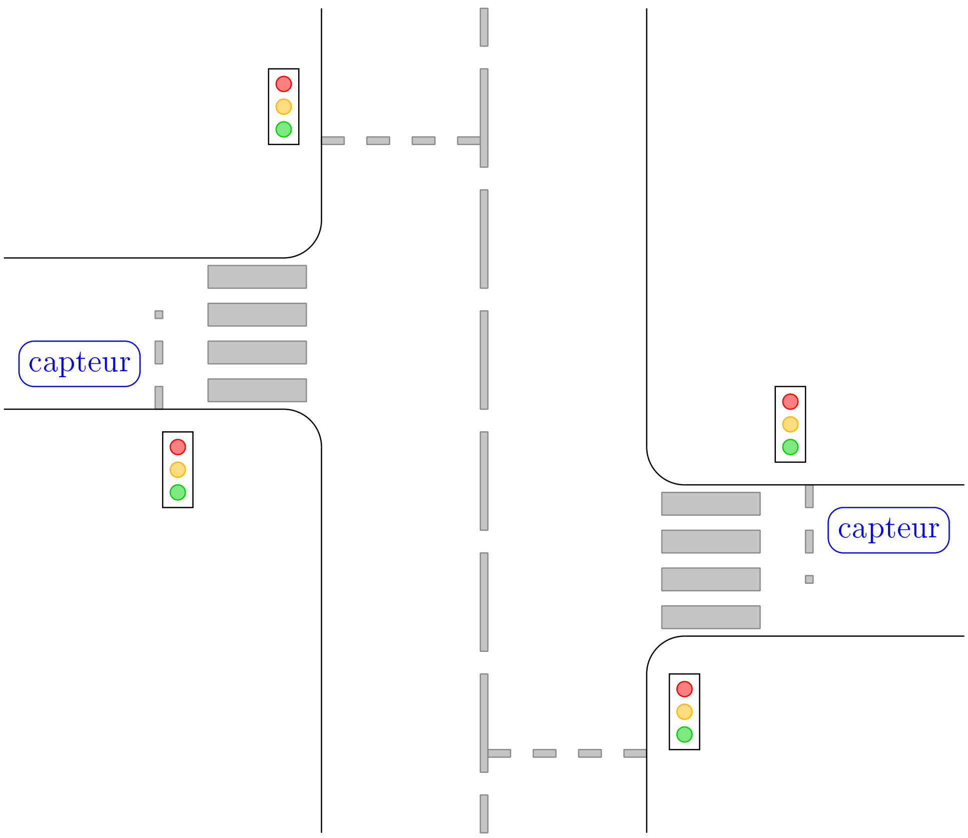

FIG. 5.4.: Feux de signalisation à un carrefour.

ne correspond pas du tout à la propriété que nous souhaitons exprimer. En effet, en l'absence d'hypothèses sur l'environnement, rien ne dit que la formule AG AF `capteur_secondaire_activé` est vraie (nous souhaitons justement l'utiliser pour faire une hypothèse sur le comportement des entrées). Il est possible de trouver un contre-exemple : un cas où le moteur est déclenché et où le capteur du bas reste indéfiniment sec. Dans ce cas, la formule est vraie quelle que soit la couleur du feu principal !

Le problème vient du fait que, en CTL, tout quantificateur sur les chemins doit être combiné avec un combinateur d'états. Par conséquent, la formule CTL ci-dessus comporte deux quantifications : l'hypothèse et la conclusion ne parlent pas du même chemin. Au contraire, dans la formule LTL, l'hypothèse et la conclusion concernent le même chemin.

Il est possible, en CTL, d'écrire des formules qui expriment une propriété plus forte, mais la propriété donnée en exemple dans cette section ne peut pas être exprimée directement en CTL. Cela nous permet de conclure que CTL n'est pas « plus grande » que LTL. De ce point, et de celui soulevé à la question précédente, il résulte que les logiques LTL et CTL ne sont pas comparables.

5.2.5. Quelques propriétés usuelles écrites en logique temporelle

5.2.5.1. Introduction

Nous donnons dans cette section un rapide aperçu d'une utilisation possible des logiques temporelles pour la vérification de propriétés. Nous présentons, à titre d'exemple, un ensemble de propriétés que nous utiliserons couramment. Certaines sont des propriétés générales, plus ou moins applicables à tout système ; d'autres sont plus spécifiques au domaine des API.

Ce travail a été effectué en commun avec les membres du projet Vulcain et reprend pour partie les résultats du rapport final du projet [CCR01].

5.2.5.2. Programmes attentifs

Notion de fin de cycle L'écriture, entre autres, de la propriété « attentif » nécessite la définition de la proposition fdc. Celle-ci est vraie au moment où le programme considéré a atteint la fin de cycle (la fin de son évaluation), et fausse partout ailleurs. La propriété fdc dépend donc du langage dans lequel le programme vérifié est écrit.

Dans le cas de la sémantique directe, la fin de cycle correspond aux propositions suivantes :

- $m = k + 1$ (m est le compteur de ligne) pour un programme IL_0 de taille k ;
- $c = \$$ (c est la position courante) pour un programme ST_0.

Dans le cas de la sémantique modulaire, les états de fin de cycle sont ceux étiquetés par la proposition fdc.

Propriété On appelle programme *attentif* un programme dont chaque exécution passe infiniment souvent par la fin de cycle. Cela signifie qu'il n'existe aucune exécution du programme qui se mette à tourner en boucle sans redonner la main à l'interpréteur principal de l'API[7]. C'est, du point de vue de la sûreté, une propriété fondamentale de tout programme d'API. En effet, un programme non attentif (par exemple à cause d'une boucle infinie) pourrait engendrer un blocage du processus. Si fdc est la fin de cycle, le caractère attentif d'un programme s'énonce en logique LTL comme suit :

$$\mathsf{G}\ \mathsf{F}\ fdc$$

Cette propriété se lit : « pour toute exécution du programme, il est toujours vrai que fdc va inévitablement se produire ». Cela signifie que fdc se produit infiniment souvent : lorsque fdc est vraie, il est aussi vrai qu'elle va se produire à nouveau, et ainsi de suite.

Cette propriété s'énonce, de manière équivalente, en CTL, par la formule $\mathsf{AG}\ \mathsf{AF}\ fdc$.

5.2.5.3. Vivacité

La vivacité consiste à exprimer le fait qu'un événement se produit consécutivement à un événement déclencheur, et que cela est inévitable. Cette propriété s'énonce par la formule

$$\mathsf{G}\ (\textit{déclencheur} \Rightarrow \mathsf{F}\ \textit{événement})$$

Cette propriété se lit : « pour tout état de toute exécution du programme, si *déclencheur* est vrai, *événement* finit par être vrai. ».

On peut choisir de restreindre la visibilité des événements aux seuls états observables (les états de fin de cycle), au moyen de la formule

$$\mathsf{G}\ (\textit{déclencheur} \Rightarrow \mathsf{F}\ (fdc \Rightarrow \textit{événement}))$$

Cette propriété se lit : « pour tout état de toute exécution du programme, si *déclencheur* est vrai, *événement* finit par être vrai dans un état de fin de cycle. ».

5.2.5.4. Atteignabilité répétée : cas du code mort

Propriété Le code mort d'un programme est une partie du programme qui n'est jamais exécutée. La détection de code mort consiste à vérifier que toute partie du code du programme est atteignable. Nous utilisons pour cela une formule CTL d'« atteignabilité répétée » :

$$\mathsf{AG}\ \mathsf{EF}\ but$$

En vérité, cette propriété est un peu plus forte que la propriété d'absence de code mort : elle énonce que « depuis tout état de toute exécution, il est possible de trouver un chemin qui mène à *but* ». La propriété *but* peut être une propriété quelconque, en particulier une propriété qui marque le passage par une partie de code du programme.

Par exemple, on peut énoncer que la ligne de code n° i d'un programme IL_0 est toujours atteignable, par la formule $\mathsf{AG}\ \mathsf{EF}\ (m = i)$.

[7] Ou, de manière plus positive, toute exécution finit toujours par redonner la main à l'interpréteur.

5.2.5.5. Exclusion mutuelle

De nombreux systèmes possèdent des variables de sortie qui, pour des raisons de sécurité, ne doivent jamais être mises à `TRUE` simultanément. Les exclusions mutuelles sont des propriétés de sûreté. Soient s_1 et s_2 deux sorties d'un programme (le nombre peut être aussi grand qu'on le désire). L'exclusion mutuelle des sorties s_1 et s_2 s'écrit comme suit en LTL :

$$\mathsf{G}\ \neg(s_1 \wedge s_2)$$

Elle se lit : « il est toujours vrai que l'on n'a jamais s_1 et s_2 vraies dans le même état ». Dans le cas de l'exécution d'un programme d'API, il n'est pas nécessaire que des formules portant sur des sorties du programme soient vraies à tout instant. En effet, il est possible que les variables correspondantes ne vérifient pas la propriété au cours de l'exécution du programme. Mais il importe qu'elles le vérifient à la fin du cycle, plus précisément lors de l'écriture des sorties.

Par exemple, dans le cas de l'exclusion mutuelle, les variables s_1 et s_2 peuvent être vraies simultanément à certains moments de l'exécution du programme. Cela n'importe pas puisque, les valeurs n'étant pas encore transmises aux lignes de sortie, aucun risque réel ne se présente. Cela peut d'ailleurs être rendu nécessaire par nécessité de programmation (p. ex. les variables ont la même valeur pendant les quelques instructions IL que nécessite l'affectation de l'une par rapport à l'autre).

Si la propriété qu'on souhaite exprimer fait appel à des variables de sortie, il n'est opportun de contrôler leur exclusion mutuelle qu'au moment de la fin de cycle. La formule devient alors :

$$\mathsf{G}\ (fdc \Rightarrow \neg(s_1 \wedge s_2))$$

Cette formule est différente de la précédente, dans la mesure où, si fdc n'est pas vraie, alors s_1 et s_2 peuvent être à `TRUE` simultanément.

5.2.5.6. Comportement des variables

Dans cette section, nous montrons comment utiliser des formules de logique temporelle pour décrire le comportement des variables du système. Parfois, il s'agira d'émettre une hypothèse sur le comportement de variables d'entrée[8] ; parfois, il s'agira de décrire le comportement d'une variable interne ou de sortie. Nous dressons ci-dessous un inventaire de certains comportements fréquemment observés.

Valeur constante On décrit le fait que la variable x est toujours vraie par la formule

$$\mathsf{G}\ x$$

À l'inverse, la formule $\mathsf{G}\ \neg x$ exprime le fait que la variable x est toujours fausse.

[8] Décrire le comportement des variables d'entrée permet de donner des informations sur l'environnement et sur la partie opérative (le ou les appareils réels, commandés par le programme).

Immobilisation Le fait qu'une variable x varie, puis s'immobilise indéfiniment à `TRUE` ou à `FALSE`, peut s'écrire sous la forme

$$\mathsf{F}\,(\mathsf{G}\,x \vee \mathsf{G}\,\neg x)$$

Au contraire, le fait qu'une variable soit sans cesse modifiée s'énonce comme suit :

$$\mathsf{G}\,(\mathsf{F}\,x \wedge \mathsf{F}\,\neg x)$$

De même que la propriété des programmes attentifs, celle-ci énonce que x et $\neg x$ se produisent infiniment souvent.

Front montant, front descendant Un front montant, ou un front descendant, est caractérisé par le changement de valeur d'une variable. On représente un front montant sur la variable x par la formule

$$\neg x \wedge \mathsf{X}\,x$$

Cela signifie que la transition suivante verra un front montant sur la valeur de x. Nous pouvons également exprimer le fait qu'un front montant va se produire, mais sans préciser à quelle transition :

$$\neg x \wedge \mathsf{F}\,x$$

Nous pouvons également décrire le fait qu'un front montant se produira d'ici la prochaine fin de cycle :

$$\neg x \wedge \mathsf{X}\,(\neg fdc\ \mathsf{U}\ (fdc \wedge x))$$

(le X permet de conserver la validité de la formule dans le cas où on l'applique à partir d'un état de fin de cycle).

Les fronts descendants se représentent de manière duale, par exemple par

$$x \wedge \mathsf{X}\,\neg x$$

5.2.6. Conclusion

Nous avons présenté, dans cette section, les deux logiques temporelles LTL et CTL, ainsi que des exemples de leur utilisation. Les deux logiques offrent un pouvoir d'expression suffisamment riche pour permettre, en pratique, d'énoncer la plupart des propriétés pertinentes portant sur un système automatisé.

Du point de vue du model-checking, il existe des algorithmes relativement performants pour chacune des deux logiques. Le model-checking de LTL (notamment décrit par [Var96]) s'effectue en temps linéaire en la taille du système de transitions, et exponentiel en la taille de la formule (il est donc efficace pour des formules de taille réduite) ; le model-checking de CTL [CES86], plus performant, s'effectue en temps linéaire en la taille du système de transitions et de la formule.

Par la suite, nous verrons des exemples d'utilisation concrète des logiques temporelles dans le cadre de la vérification par model-checking.

5.3. Model-checking

5.3.1. Problème général

Étant donné le modèle A d'un système (programme, système automatique, protocole...) et une propriété φ, nous souhaitons établir si le modèle A vérifie φ, ce que nous notons

$$A \models \varphi$$

Le problème du model-checking (décrit notamment dans [CGP99, SBB$^+$99]) revient à répondre à la question : « le modèle A vérifie-t'il la propriété φ ? ».

Le modèle est formel, c'est-à-dire qu'il est spécifié mathématiquement, et représente le comportement du système réel qu'il modélise (ou une partie de ce comportement, de manière plus ou moins abstraite). De même, les propriétés sont spécifiées formellement.

Il existe plusieurs formalismes pour représenter les modèles : réseaux de Petri (p. ex. Design/CPN [Met]), systèmes de transitions temporisés (Kronos [Yov97], Uppaal [LPY97]), automates communicant par canaux (Spin [Hol91])... Pour sa part, SMV utilise des structures de Kripke, c'est-à-dire des systèmes de transitions dont les états peuvent être étiquetés par des propositions atomiques. Pour l'écriture de propriétés, les logiques temporelles sont assez largement utilisées, de manière plus ou moins directe, en raison notamment de l'existence d'algorithmes performants pour en effectuer le model-checking.

L'efficacité des algorithmes est primordiale ; en effet, la résolution d'un problème de model-checking suppose une exploration exhaustive des états accessibles du système étudié. Si on considère, comme c'est par exemple le cas de la sémantique décrite à la section 4.2.3 page 49, que les états sont constitués d'une valuation des variables, il apparaît clairement que le nombre d'états du système est exponentiel par rapport au nombre de variables ; cela rend hors de portée la vérification exhaustive de ces états.

Nous verrons que SMV, par l'utilisation de méthodes puissantes de représentation symbolique, permet d'effectuer la vérification de systèmes de très grande taille [JEK$^+$92, CGL92]. Cela est également dû à l'existence d'algorithmes performants pour le model-checking de propriétés écrites en logique temporelle, y compris pour les systèmes représentés symboliquement.

5.3.2. Model-checking symbolique

5.3.2.1. Principes

Nous avons vu, à la section 5.1 page 91, qu'il existait des algorithmes performants pour le model-checking des formules de logique temporelle. Plus précisément, ces algorithmes sont généralement linéaires en la taille du système de transitions (nombre d'états, de transitions). Cela garantit l'obtention du résultat dans un temps très réduit, sous réserve que l'automate soit de taille raisonnable, ce qui est hélas rarement le cas. En effet, si on imagine, par exemple, que les états du système sont formés par une valuation de ses variables, il apparaît clairement que le nombre d'états est exponentiel par rapport au nombre de variables. Nous nous trouvons confrontés à ce qu'on appelle l'« explosion du nombre d'états ».

Quelle que soit la performance des algorithmes, la vérification devient dès lors hors de portée, du moins tant que l'on décide d'énumérer et de représenter explicitement en mémoire tous les états de l'automate. Le model-checking symbolique permet, à l'aide d'algorithmes simples et également performants, de vérifier de bien plus grands systèmes, à partir du moment où l'on dispose d'une méthode de représentation symbolique du système de transitions, c'est-à-dire une méthode permettant de le représenter sans devoir énumérer l'ensemble des états. De telles méthodes existent ; la plus couramment utilisée dans le model-checking (et en particulier dans SMV) est celle des OBDD.

5.3.2.2. Les OBDD

Principes Les OBDD (*ordered binary decision diagrams*) sont des structures de données permettant de représenter des formules booléennes, que nous pouvons représenter graphiquement. C'est Bryant qui a popularisé leur utilisation [Bry86, Bry92], et leur application au model-checking a été proposée par trois équipes différentes [CBM90, JEK+92, Pix92]. Cette méthode est efficace, car elle propose des règles de simplification, qui permettent :
- de garantir l'existence, après l'application de ces règles et pour un ordre donné sur les variables, d'une forme canonique, qui permet de comparer simplement deux formules booléennes différentes entre elles ;
- de conserver les opérations booléennes courantes : à partir d'une ou plusieurs formules, représentées sous forme simplifiée, il existe des algorithmes simples permettant d'appliquer toutes les opérations booléennes courantes (*et*, *ou*, *non*...) et d'obtenir une représentation simplifiée de la formule résultante.

Nous ne détaillerons pas les opérations de simplification sur les OBDD (dont on trouvera des descriptions simples et claires dans la littérature, par exemple [SBB+99, McM93]), mais nous les illustrons aux figures 5.5 et 5.6 page suivante. Les figures se lisent, de haut en bas, en suivant la flèche correspondant à la valeur de la variable figurant dans chaque nœud rencontré. Au bout du chemin suivi, la valeur de l'expression est donnée par la valeur située dans la feuille. La figure 5.5 montre la représentation graphique d'une formule booléenne sous la forme d'un arbre de décision ; la figure 5.6 montre la même formule, après simplification, sous la forme d'un OBDD.

Ordre des variables Au vu de l'exemple précédent, on remarque qu'un ordre a été donné sur les variables (ordre dans lequel les décisions booléennes sont prises). La construction d'un OBDD est effectivement dépendante d'un tel choix (d'où le *ordered* de son nom). Cependant, il faut savoir que l'ordre dans lequel les choix sont faits n'est pas sans influence sur la complexité de la représentation. Par exemple, la figure 5.7 page 107 montre une formule booléenne, paramétrée par un entier n, représentée par deux OBDD différents, en fonction de l'ordre choisi pour les variables. Une des deux représentations est de taille linéaire, alors que l'autre est exponentielle !

Model-checking Les OBDD peuvent être utilisés comme représentation symbolique d'un système de transitions, car la relation de transition peut être codée par une formule boo-

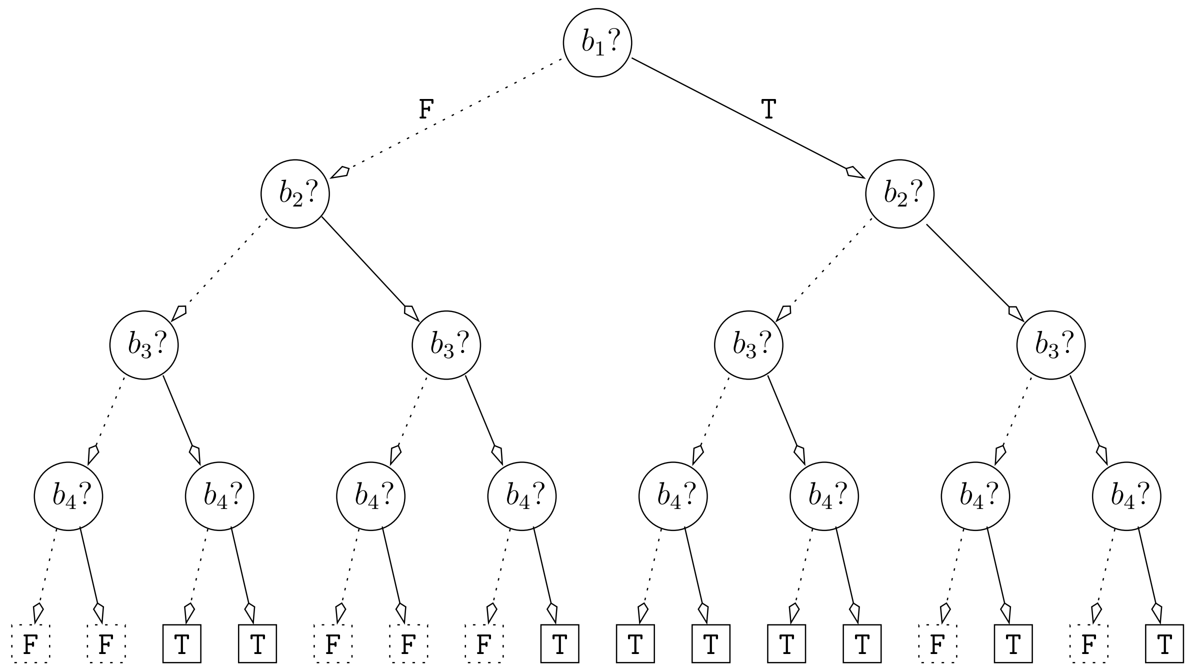

FIG. 5.5.: Représentation de la formule $(b_1 \vee b_3) \wedge (b_2 \Rightarrow b_4)$ sous la forme d'un arbre de décision, sans simplification. Les symboles `T` et `F` représentent les constantes `TRUE` et `FALSE`.

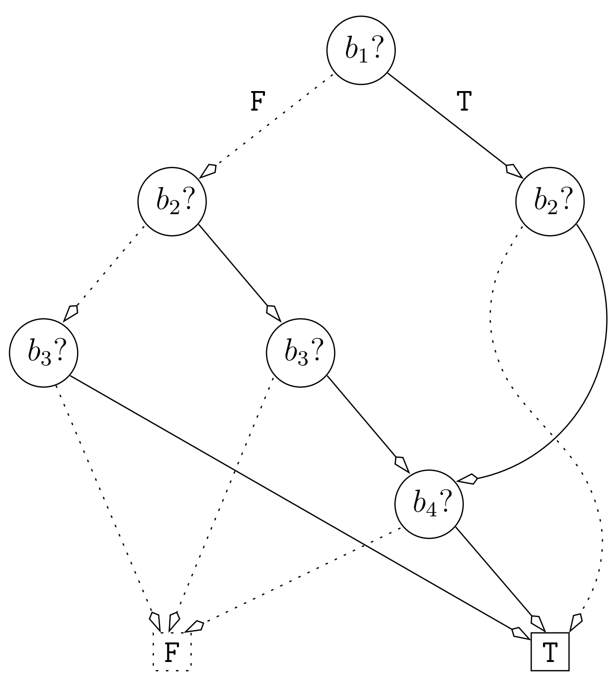

FIG. 5.6.: Représentation de la même formule qu'à la figure 5.5, sous la forme d'un OBDD.

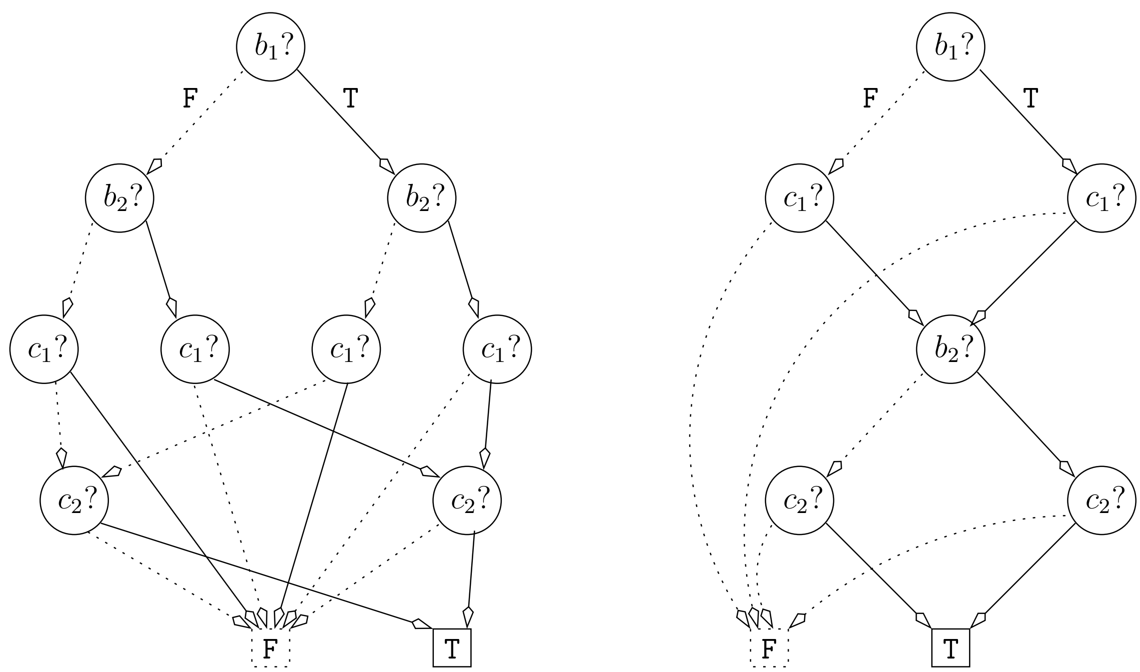

FIG. 5.7.: Représentation de la formule $(b_1 \Leftrightarrow c_1) \wedge \ldots \wedge (b_n \Leftrightarrow c_n)$ dans le cas $n = 2$, par deux OBDD ; les deux OBDD ne diffèrent que par l'ordre donné sur les variables. En fonction de n, la taille du premier est exponentielle, celle du deuxième est linéaire.

léenne. Un tel codage est illustré par sa mise en œuvre dans SMV, décrite à la section 5.4.2 page ci-contre.

Nous avons vu que, du point de vue de la complexité du model-checking, CTL possède un avantage sur LTL. Cela est principalement dû au fait que CTL ne permet d'écrire que des formules sur les états et pas sur les chemins, et que les états sont plus simples à traiter algorithmiquement que les chemins. Cet avantage se creuse d'autant plus dans le cas du model-checking symbolique, où l'utilisation de CTL est nettement plus efficace. Nous verrons cependant, à la section 5.4.3.3 page 114, qu'il existe des méthodes permettant d'utiliser des algorithmes de model-checking CTL pour vérifier des formules écrites en LTL.

Conclusion Les OBDD permettent d'effectuer une représentation symbolique d'un système de transitions. La simplification qu'ils apportent est variable, et dépend de plusieurs facteurs, parmi lesquels un ordre sur les variables, dont le choix revient à l'« utilisateur ». Il peut, dans certains cas, être possible d'inférer un ordre efficace sur les variables, mais la question du choix de l'ordre est complexe, et il n'existe pas de méthode systématique efficace donnant un ordre optimal. Selon le choix effectué, la même vérification, de triviale, peut devenir infaisable, et réciproquement.

Cependant, et malgré ces limites, l'apport des OBDD est considérable en pratique, et s'avère fructueux dans de nombreux cas, en permettant la vérification de systèmes de très grande taille.

5.4. SMV

5.4.1. Introduction

Le logiciel SMV est un outil de model-checking symbolique performant, que nous décrivons dans cette section. Nous décrirons en particulier le langage d'entrée de SMV (suivant l'exemple de son manuel d'utilisation, nous l'appellerons le langage SMV), que nous utiliserons pour donner une spécification, automatiquement déduite de la sémantique opérationnelle, des programmes écrits dans les langages IL et ST.

Les deux SMV Il existe en réalité deux logiciels de model-checking portant le nom SMV :

CMU SMV, réalisé à Carnegie Mellon[9] par K. L. McMillan. C'est le logiciel décrit dans le livre [McM93]. Il effectue le model-checking de programmes ou protocoles décrits sous la forme d'un système de transitions, au moyen d'un langage dédié. Les propriétés sont décrites en CTL. Malgré le départ de son auteur, le logiciel est toujours maintenu ;

Cadence SMV, réalisé à Cadence Berkeley Labs[10] par le même auteur. Les principes sont les mêmes, avec quelques modifications (voir la section 5.4.3.4 page 116). Par

[9] CMU SMV peut être téléchargé à l'adresse `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[10] Cadence SMV peut être téléchargé à l'adresse `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

rapport à son cousin, Cadence SMV propose, entre autres nouveautés, des méthodes heuristiques pour détecter automatiquement un ordre des variables efficace (voir la section 5.4.2.7 page 113) et une méthode permettant d'effectuer des abstractions du système par couches. Le langage de spécification de Cadence SMV est différent de celui de CMU SMV. Selon sa documentation [McM], les propriétés s'écrivent en LTL, mais elles sont en fait traduites par le logiciel dans un modèle CTL avec contraintes d'équité (voir les sections 5.4.3.2 et 5.4.3.3 page 113). Le model-checking s'effectue donc en CTL. Il est malgré tout possible, bien que cela ne soit pas documenté, de spécifier des formules en CTL.

Dans la suite de ce document, c'est le logiciel Cadence SMV que nous utiliserons ; c'est son langage de spécification que nous décrivons dans cette section.

Cadence SMV : les modules Parmi les éléments significatifs du langage SMV que nous utilisons, les modules ont une importance fondamentale, car ils permettent de coder naturellement la synchronisation des automates de la sémantique opérationnelle modulaire, en conservant sa philosophie, à savoir :
- la possibilité de définir chaque élément de langage une fois pour toutes, puis de l'instancier pour le réutiliser autant que nécessaire ;
- une interface unifiée pour chaque type d'élément, à base de variables assimilables à des messages.

Ce chapitre ne vise pas à donner une description complète du langage SMV. On se reportera à [McM] pour une telle description. Nous présentons ici quelques principes fondamentaux, et les structures principales de SMV, qui permettront à l'utilisateur intéressé de l'aborder plus facilement. Au chapitre suivant, nous expliquerons comment exploiter le langage pour coder automatiquement une spécification à partir d'un programme IL ou ST.

5.4.2. Caractéristiques de SMV

5.4.2.1. Principes du langage

SMV est un logiciel permettant d'effectuer la vérification, par model-checking, d'un système décrit sous la forme d'un système de transitions. Le système à vérifier est spécifié au moyen d'un langage que nous appelons le langage SMV. À chaque spécification en SMV, on ajoute un ensemble de formules de logique temporelle, dont SMV effectue la vérification. Si l'une de ces propriétés est fausse, SMV s'arrête et donne une trace (une exécution) en guise de contre-exemple, qui peut être utilisée pour trouver l'erreur qui a causé l'échec.

Avant de présenter les principales caractéristiques du langage SMV, il convient de se rappeler qu'il permet de décrire un système de transitions, par la donnée de sa relation de transition. C'est en effet celle-ci qui est stockée dans le modèle mémoire interne de SMV, sous la forme d'une formule booléenne (représentée avec des OBDD), dans laquelle, pour chaque variable du système, intervient un couple de valeurs : la valeur de la variable avant la transition, et la valeur après la transition. Dans le langage SMV, si x est une variable déclarée, x désigne la valeur de la variable avant la transition et `next`(x) la valeur après la transition.

Les états du système de transition sont composés d'une valuation de toutes les variables du système spécifié. Cette forme étant très proche du modèle formel de la sémantique directe (décrite à la section 4.2 page 49), nous utiliserons celui-ci pour illustrer la définition du langage. Nous supposons donc que le langage SMV code un système de transitions de la forme $\langle Q, \rightarrow, I\rangle$, où Q est l'ensemble des états, $\rightarrow$ la relation de transition et $I \subseteq Q$ l'ensemble des état initiaux. Chaque état $q \in Q$ du système est une valuation des variables. On désignera donc par $q(a)$ la valeur de la variable a dans l'état q.

Étant donnés deux états du système, la formule booléenne qui représente la relation de transition permet de déterminer si une transition existe du premier état vers le second. Pour ce faire, on donne à chaque variable x sa valeur dans l'état de départ et à chaque `next`(x) sa valeur dans l'état d'arrivée, puis on applique la formule booléenne : si elle donne un résultat vrai, il existe une transition du premier au second état, si elle est fausse, il n'en existe pas. Par exemple, une formule toujours vraie correspond à une relation totale : tous les états sont directement reliés entre eux par des transitions. Inversement, une formule toujours fausse représente une relation de transition vide.

Le langage SMV est donc un langage déclaratif. Il nous permet de décrire un système en fonction de l'évolution de ses variables au cours du temps. La formule booléenne qui représente la relation de transition est automatiquement déduite, par le compilateur, de la spécification en langage SMV.

5.4.2.2. Opérateur d'affectation

L'opérateur de base du langage est l'opérateur d'affectation (`:=`). Cet opérateur permet de spécifier des liens entre des variables. Par exemple :

```
next(x) := y;
```

Cette ligne de code signifie que la variable `x` doit toujours avoir la valeur de la variable `y` « à l'étape suivante ». Pour bien comprendre ce que cela signifie, nous pouvons imaginer la formule booléenne qui en découle : `next(x)` $=$ `y`.

Cela revient à spécifier le système de la manière suivante : pour tous $q, q' \in Q$, $q \rightarrow q' \Rightarrow q'(\mathtt{x}) = q(\mathtt{y})$. Autrement dit, chaque transition doit être telle que la valeur de `x` après la transition soit égale à la valeur de `y` avant la transition. Écrire cette ligne de code revient à éliminer toutes les transitions ne correspondant pas à ce critère. Un système comportant une spécification vide est un système dont le comportement est entièrement aléatoire (la formule qui lui correspond est la tautologie `TRUE`). Spécifier le comportement d'une variable revient à retirer des transitions. Inversement, ne pas spécifier le comportement d'une variable revient à lui autoriser toutes les évolutions possibles.

Il est possible de définir une affectation non déterministe, de la manière suivante :

```
next(x) := {2, 4};
```

Cela signifie, en l'absence d'autre restriction, qu'il existe une transition entre n'importe quel état, d'une part, et tout état dans lequel la variable `x` vaut 2 ou 4, d'autre part. Cela équivaut à la formule booléenne $\mathtt{next}(\mathtt{x}) = 2 \vee \mathtt{next}(\mathtt{x}) = 4$.

5.4.2.3. Valeurs initiales

Une valeur initiale peut être spécifiée pour une variable à l'aide du mot-clé `init` :

```
init(x) := 0;
```

La ligne qui précède signifie que, si x est une variable booléenne, pour tout $q \in Q$, $q \in I \Rightarrow q(\mathtt{x}) = \mathtt{FALSE}$[11]. Autrement dit, les états initiaux sont tous tels que `x` y vaut `0`.

5.4.2.4. Ordre des déclarations

Il est dès lors facile de remarquer que l'ordre des déclarations n'importe pas : les affectations peuvent être décrites dans n'importe quel ordre. On pourrait cependant imaginer le problème que poserait le cas où la même variable serait affectée deux fois dans la même spécification SMV. Les concepteurs de SMV ont choisi d'interdire cette situation, qui est traitée comme une erreur par le compilateur SMV. Chaque variable ne doit donc être affectée qu'une unique fois.

5.4.2.5. Opérateurs conditionnels

Le langage offre la possibilité d'utiliser des expressions conditionnelles. À cet effet, un mot-clé `case` est défini dans le langage :

```
next (x) := case
            {
              y = 0   : 0;
              default : x;
            };
```

Le code qui précède décrit un système tel que tout état dans lequel la variable `y` vaut `FALSE` est suivi par un état dans lequel la variable `x` vaut `FALSE`. De plus, tout état dans lequel la variable `y` ne vaut pas `FALSE` est suivi par un état dans lequel la valeur de la variable `x` est conservée, ce que nous pouvons écrire de la manière suivante :

$$\forall q, q' \in Q,\ q \rightarrow q' \Rightarrow \begin{cases} q'(\mathtt{x}) = \mathtt{FALSE} \text{ si } q(\mathtt{y}) = \mathtt{FALSE} \\ q'(\mathtt{x}) = q(\mathtt{x}) \text{ sinon} \end{cases}$$

La règle de lecture du `case` est la suivante : les conditions sont lues du haut vers le bas ; est retenue la première définition dont la condition est vraie. La condition `default`, optionnelle, est toujours vraie. Si, pour une valuation particulière des variables, aucune condition

[11] SMV désigne les constantes booléennes `TRUE` et `FALSE` par les entiers `1` et `0`.

n'est vraie, alors le comportement de la variable n'est pas spécifié, et par conséquent toutes les évolutions lui sont possibles, de façon non déterministe.

Le code ci-dessus peut également être écrit de manière plus condensée, en utilisant le mot-clé `switch`, qui permet de quantifier sur une variable :

```
next (x) := switch (y)
              {
                0       : 0;
                default : x;
              };
```

5.4.2.6. Notion de module

Le langage SMV propose une structure de module permettant de définir des éléments réutilisables. Un module SMV est une structure de données comprenant un ensemble de variables, dont le comportement peut être spécifié au moyen des opérateurs habituels. Ce qui suit est un exemple de module valide :

```
module demo_module (i, o)
  {
    input i : boolean;
    output o : boolean;
    init (o) := 0;
    next (o) := i;
  }
```

Un module est un type de donnée. Il est instancié en déclarant une variable ayant ce module pour type. Par exemple, nous déclarons comme suit une variable `module_1`, instance du module `demo_module` :

```
var_i : boolean;
var_o : boolean;
module_1 : demo_module (var_i, var_o);
```

Les variables `i` et `o`, définies à l'instanciation, sont accessibles en utilisant la syntaxe `module_1.i` et `module_1.o`. Ce sont des variables locales au module. Elles sont placées en argument, de manière à établir les correspondances entre variables deux à deux. La nature de ces correspondances est précisée par les mots-clés `input` et `output`, qui ont pour effet d'ajouter des liaisons implicites entre variables pour chaque module instancié. Ainsi, les liaisons suivantes sont implicitement ajoutées par SMV lors de l'instanciation du module :

```
module_1.i := var_i;
var_o := module_1.o;
```

Conformément aux règles de liaison définies par SMV (chaque variable ne peut être affectée qu'au plus une fois), il n'est donc pas autorisé de spécifier une affectation de la variable `var_o`, dès lors qu'elle est liée à une variable définie par `output` dans un module, de même qu'il est pas autorisé de spécifier une affectation d'une variable définie comme `input` dans un module.

5.4.2.7. OBDD et ordre des variables

SMV représente en mémoire la relation de transition par des OBDD. Comme nous l'avons vu à la section 5.3.2.2 page 105, ce mode de représentation est très efficace, mais dépend considérablement d'un ordre fixé sur les variables. Cadence SMV propose des méthodes heuristiques pour inférer un bon ordre sur les variables. Ces méthodes, non précisées par l'auteur, se révèlent à l'usage très efficaces ; il est néanmoins possible, en cas d'échec, de spécifier son propre ordre sur les variables.

5.4.3. SMV et les logiques temporelles

5.4.3.1. Les logiques acceptées par SMV

Nous avons vu, à la section 5.2.4 page 97, que les logiques CTL et LTL possédaient des pouvoirs d'expression distincts. La pratique a permis de déterminer que, en règle générale, LTL est mieux adaptée à la description des comportements attendus des systèmes. D'autre part, les formules de grande taille écrites en CTL sont souvent difficiles à interpréter ; la présence obligatoire de plusieurs quantificateurs sur les chemins peut rendre les formules contre-intuitives (voir l'exemple des hypothèses dynamiques, section 5.2.4.2 page 98).

Du point de vue de la vérification, cependant, CTL comporte un avantage considérable : le model-checking symbolique de CTL est beaucoup plus efficace que celui de LTL. Il est, de plus, possible d'étendre la logique CTL par des contraintes d'équité (voir la section suivante). Cette extension permet d'agrandir le pouvoir d'expression de CTL, sans augmenter considérablement la complexité du model-checking.

Les concepteurs de Cadence SMV ont choisi de faire un compromis entre les deux : SMV exprime les propriétés, de manière interne, en CTL avec contraintes d'équité. Il offre, de plus, la possibilité d'écrire les propriétés aussi bien en LTL qu'en CTL. Pour ce faire, il utilise une méthode permettant de vérifier des formules LTL à l'aide d'algorithmes de model-checking CTL avec contraintes d'équité.

5.4.3.2. CTL et équité

SMV étend le pouvoir d'expression de CTL par l'utilisation de contraintes d'équité. Cela est différent de ce qu'on appelle usuellement CTL avec équité, qui consiste à ajouter à la logique un combinateur sur les chemins (ce combinateur est noté $\overset{\infty}{\mathsf{F}}$ et est équivalent à GF : *infiniment souvent*).

Pour SMV, il s'agit de l'ajout de contraintes au modèle. Une telle contrainte est une formule φ de logique temporelle ; elle est interprétée par : « dans tous les cas, le modèle vérifie la formule φ infiniment souvent » (autrement dit, il n'existe pas d'exécution infinie dans laquelle φ, au bout d'un moment, cesse d'être vérifiée pour toujours).

Chaque contrainte ajoutée permet donc de restreindre le parcours des exécutions, en éliminant les exécutions non équitables. Cette fonctionnalité est utile, par exemple pour ajouter des hypothèses de vivacité dans le modèle. Ainsi, nous pouvons utiliser les contraintes d'équité pour résoudre le problème des hypothèses dynamiques (exposé à la section 5.2.4.2 page 98). En reprenant l'exemple de la signalisation à un carrefour, la propriété énoncée à la section 5.2.4.2 s'énonce par la formule de vivacité suivante, que nous écrivons en CTL :

$$\mathsf{AG}\ \mathsf{AF}\ \texttt{feu\_principal\_rouge}$$

En soi, la formule n'est pas vérifiée. Nous pouvons vérifier notre propriété en ajoutant une contrainte d'équité sur la formule suivante :

`capteur_secondaire_activé`

Ainsi, on ne retient que les exécutions dans lesquelles le capteur de la route secondaire est activé de temps à autre (mais ne cesse jamais indéfiniment de l'être), et nous pouvons vérifier la propriété portant sur le comportement des feux dans cette hypothèse.

Cette extension permet d'augmenter le pouvoir d'expression de la logique, sans restreindre la complexité du model-checking. Elle a le défaut d'être peu souple (par rapport, par exemple, à l'ajout d'un opérateur d'équité), mais elle rend possible l'utilisation de formules LTL dans un model-checker CTL.

5.4.3.3. De LTL à CTL

Méthode générale La méthode permettant de vérifier des formules LTL à l'aide d'un model-checker CTL+équité repose sur un résultat de Vardi et Wolper [VW86], qui propose un codage de la formule LTL de départ sous la forme d'un système de transitions. À partir de ce système de transitions, Clarke *et al* [CGH97] proposent une mise en œuvre de la vérification, exploitant la technique du model-checking symbolique (en utilisant SMV). La méthode générale repose sur le principe selon lequel on peut affiner la vérification d'une propriété en éliminant des chemins du système. SMV propose deux moyens d'éliminer des chemins :

- ajouter des contraintes à la relation de transition (c'est-à-dire augmenter la formule booléenne qui le représente, par l'ajout d'une ou plusieurs clauses) ;
- ajouter des contraintes d'équité (voir la section précédente).

Dans un premier temps, la méthode présentée dans [CGH97] utilise un résultat de [VW86], qui permet de coder la formule LTL à vérifier, indépendamment du contexte, sous la forme d'un système de transitions étiqueté (aussi appelé structure de Kripke). Les étiquettes, placées sur les états, correspondent aux propositions atomiques de la formule, et à d'autres sous-formules LTL (par exemple une étiquette Xp représente $\mathsf{X}\ p$, si p est une proposition apparaissant dans la formule de départ). Le but de cette représentation est de définir une structure de Kripke « cohérente », c'est-à-dire dont toute étiquette est vérifiée : par exemple, si un état est étiqueté par $X\varphi$, tous ses successeurs sont étiquetés par φ. Une telle cohérence s'obtient, avant tout, en définissant judicieusement la relation de transition.

Une telle définition directe de la relation de transition est simple, c'est-à-dire de faible complexité, et peut être mise en œuvre, sous SMV, par l'ajout de clauses à la formule représentant la relation de transitions : par exemple, l'ajout de la formule $\texttt{next}(p) \Leftrightarrow Xp$ permet de caractériser une étiquette Xp.

Cependant, nous pouvons constater que la simple caractérisation, de cette manière, de la relation de transition, ne suffit pas, car elle ne permet pas de caractériser une propriété portant sur un chemin entier : par exemple, dans le cas d'une formule $a\ \mathsf{U}\ b$, une telle définition permet de valider une transition d'un état étiqueté par a à un autre état étiqueté par a, mais rien ne permet d'affirmer que cette transition appartient à un chemin qui, inévitablement, conduira à b. Si on considère une proposition atomique supplémentaire aUb (*i.e.* une variable ajoutée au système), une telle garantie s'obtient par la prise en compte d'une contrainte d'équité de la forme

$$\neg(aUb) \vee b$$

On peut interpréter cette contrainte d'équité de la manière suivante : si aUb est vraie (n'est pas fausse), b finit inévitablement par être vraie (car la contrainte d'équité est vérifiée infiniment souvent). L'ajout de telles contraintes permet de garantir la cohérence totale d'une structure de Kripke, ainsi construite à partir de toute formule LTL.

Une fois la formule complètement caractérisée sous la forme d'un système de transitions, on effectue un produit (défini spécialement pour cette méthode) entre le système représentant la formule, et le modèle du système à vérifier (lui-même représenté sous la forme d'un système de transitions). Ce produit a pour effet d'éliminer du modèle les exécutions qui ne vérifient pas la propriété.

Exemple de mise en œuvre sous SMV Selon son auteur [SMV], une méthode similaire, visiblement inspirée de la précédente, mais de manière plus étendue, est utilisée dans SMV. Supposons par exemple que nous souhaitions vérifier, avec SMV, la formule LTL $\mathsf{F}\ \mathsf{G}\ p$. Cette formule est transformée en $\mathsf{AF}\ Gp$, où Gp est définie comme une nouvelle variable, dont le comportement est caractérisé d'après la méthode décrite précédemment :

1. la relation de transition d'un système vérifiant Gp est caractérisée par la formule booléenne
$$Gp \Leftrightarrow p \wedge \texttt{next}(Gp)$$
Cette formule est ajoutée à la formule représentant la relation de transition. Comme nous l'avons vu, cela permet de caractériser simplement la relation de transition, mais ne suffit pas : chaque transition ainsi définie appartient à au moins un chemin qui vérifie la propriété $\mathsf{G}\ p$, mais il reste des chemins qui ne la vérifient pas ;
2. on ajoute une contrainte d'équité sur la formule
$$Gp \vee \neg p$$
que l'on peut lire ainsi : si Gp est faux, p finit inévitablement par être vraie.

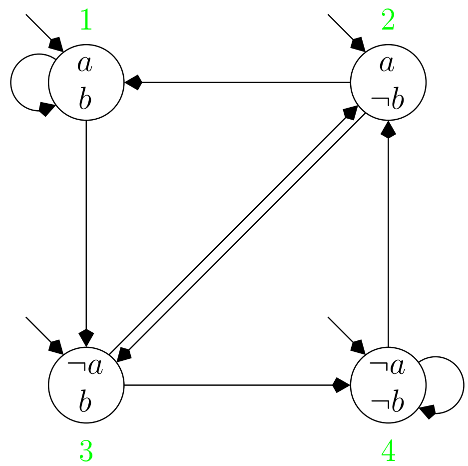

FIG. 5.8.: Système de transitions dont toutes les variables sont sur les états.

Ainsi définie, la variable Gp, placée dans les états du système à vérifier, caractérise le comportement de la formule $\mathsf{G}\ p$. Il reste alors, pour SMV, à lancer la vérification de la formule $\mathsf{AF}\ Gp$.

5.4.3.4. Pourquoi utiliser LTL ?

Comme décrit aux sections 5.2.4.2 page 98 et 5.4.3.2 page 113, la logique LTL permet d'exprimer des propriétés de manière plus naturelle, et plus facile à interpréter. L'ajout de contraintes d'équités à CTL, bien que d'une grande utilité pour permettre la vérification d'un plus grand nombre de propriétés, est une solution rarement triviale à mettre en œuvre, et manque de souplesse, notamment dans le cas de propriétés qui peuvent s'énoncer directement en LTL.

La vérification de formules CTL est rendue plus délicate encore, par la conception même de Cadence SMV. Celui-ci utilise en effet un modèle légèrement différent de celui de CMU SMV, sans que cela soit documenté (on trouvera une indication de ce fait dans les archives de la liste de diffusion du logiciel [SMV]). Dans ce dernier, les états sont composés par une valuation des variables, ce qui revient à dire que les variables sont des étiquettes sur les états. Au contraire, Cadence SMV place certaines variables, non sur les états, mais sur les transitions. Prenons l'exemple d'un système à deux variables a et b, telles que a est une variable d'entrée qui peut prendre n'importe quelle valeur à tout moment, et b est une variable de sortie qui prend la valeur de a à la transition précédente (le système serait alors défini par la seule affectation `b := next(a)`). Il n'y a aucune initialisation des variables : tous les états sont initiaux. Si nous plaçons les deux variables dans les états, nous obtenons le système de transitions représenté à la figure 5.8. Si nous plaçons la variable a dans les transitions, nous obtenons le système de transitions représenté à la figure 5.9 page ci-contre.

Cette modification du modèle a pour but d'améliorer les performances du model-checking, mais elle comporte un inconvénient. Supposons par exemple que nous souhai-

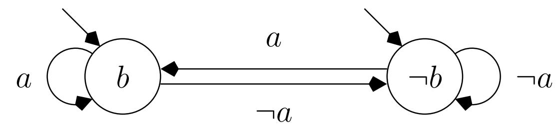

FIG. 5.9.: Système de transitions dont une variable est placée dans les transitions.

tions vérifier la propriété CTL $\mathsf{EX}\ b$, c'est-a-dire « à partir de tout état initial, il existe une transition menant à un état où b est vrai ». On voit que le système de la figure 5.8 ne vérifie pas la propriété : l'état 4 ne mène à aucun état où b est vrai[12]. Le système de la figure 5.9, quant à lui, vérifie bien la propriété.

Le changement de modèle, pour utile qu'il soit, a donc une influence sur la véracité des propriétés écrites en CTL. Cette influence est limitée aux propriétés comportant, dans une certaine configuration, le quantificateur E. Plus précisément, il existe un fragment de CTL, appelé ACTL, dont les formules ne souffrent pas de ce défaut. Les formules LTL, quant à elles, ne posent aucun problème.

Les variables placées par Cadence SMV dans les transitions sont les variables vérifiant les deux conditions suivantes :

1. elles ne sont affectées nulle part dans la spécification ;
2. elles ne figurent dans aucune formule à vérifier.

Bien que claires, la maîtrise de ces conditions est difficile à conserver dans le cas concret de la vérification d'un système en cours d'évolution : le simple ajout d'une contrainte, ou même d'une formule, peut entraîner une modification du modèle, et influer, par voie de conséquence, sur la vérification des formules CTL.

Pour cette raison, et pour la commodité d'utilisation de LTL, on préférera donc, autant que possible, exprimer les propriétés en LTL. Par ailleurs, dans le cas de Cadence SMV, seule l'utilisation de LTL est désormais documentée. Il est cependant toujours possible d'utiliser CTL lorsque c'est nécessaire (notamment pour vérifier des propriétés d'atteignabilité), avec certaines précautions.

[12]Notons au passage que la propriété $\mathsf{EX}\ b \vee \mathsf{EX}\ \neg b$ n'est pas vérifiée, ce qui est correct mais contre-intuitif.

6. Codage en SMV des programmes IL et ST

6.1. Introduction

De la sémantique opérationnelle des langages, nous déduisons une méthode permettant d'associer, à tout programme bien formé, sa spécification en SMV. Ce chapitre présente les codages de chacune des trois sémantiques présentées au chapitre 4 : sémantique directe IL, sémantique directe ST, sémantique modulaire ST. Le lien entre la sémantique de départ et son codage sous SMV est d'autant plus apparent que les modèles associés sont très proches (systèmes de transitions aux états étiquetés), et que les méthodes de codage proposées ne font pas l'objet d'optimisations particulières. Il sera possible, dans un deuxième temps, d'effectuer de telles optimisations, sous réserve de prouver leur correction.

Ce chapitre présente chacun des trois codages ; des exemples complets sont présentés au chapitre suivant.

6.2. Sémantique directe IL

6.2.1. Principes

La sémantique opérationnelle est codée presque littéralement dans le langage SMV. L'originalité du codage en SMV est que son langage oblige à caractériser l'évolution de chaque variable séparément. On déclarera une variable sous SMV pour chaque variable du modèle, évoluant de manière similaire. On retrouvera en particulier les spécificités de chaque type de variable (entrée, sortie, interne). On rappelle que les états du système sont des triplets (V, a, i). Les paramètres additionnels sont représentés de la manière suivante :

- l'accumulateur a est représenté par une variable `a` ;
- la ligne courante i est représentée par une variable `pc`.

Cette section présente en détail le codage utilisé. Ce codage a fait l'objet d'une implémentation, sous la forme d'un programme Perl relativement simple (en raison de la simplicité de l'analyse syntaxique des programmes IL_0), que l'on trouvera à l'annexe C page 185.

6.2.2. Contrôle

6.2.2.1. Continuation et cycle

La continuation est représentée, dans la sémantique opérationnelle, par un paramètre i indiquant la ligne courante. Elle est représentée dans notre codage par une variable `pc`. Si k est la taille du programme, la variable `pc` est une variable entière dont la valeur s'étend de 1 à $k+1$. Sa définition comprend une simple incrémentation à chaque transition, puis une réinitialisation en fin de cycle. La variable `pc` évolue également lors des sauts. Par exemple, si $k = 9$, si la ligne 3 comprend un saut conditionnel `JMPC l`, et que l'étiquette `l` pointe sur la ligne 7, la définition de `pc` est la suivante :

```
pc : 1 .. 10;
init (pc) := 1;
next(pc) := case
  {
    pc = 3 & a : 7;
    pc = 10 : 1;
    default : pc + 1;
  };
```

6.2.2.2. Accumulateur

La variable `a` représente l'accumulateur. Par exemple, si le programme comporte la commande `LD y` à la ligne 4, la commande `OR z` à la ligne 5, et la commande `LDN y` à la ligne 7, la variable `a` est ainsi définie :

```
next(a) := case
  {
    pc = 4  : y;
    pc = 5  : a | z;
    pc = 7  : ~y;
    default : a;
  };
```

6.2.3. Variables

À chaque variable x du modèle, on fait correspondre une variable `x` dans le codage. Ces variables évoluent de la même manière ; on retrouvera par conséquent les comportements caractéristiques des variables d'entrée et de sortie.

Les variables d'entrée n'évoluent qu'au cours de la transition d'acquisition. Par exemple, si $k = 9$, l'évolution de la variable d'entrée `y` se définit de la manière suivante :

```
next (y) := switch (pc)
  {
    10      : {0, 1};
```

```
    default : y;
  };
```

La variable y prend toutes les valeurs possibles, de façon non déterministe, lors de la transition d'acquisition ; elle conserve sa valeur le reste du temps.

Les variables autres que les entrées évoluent lorsqu'elles sont affectées dans le programme. Dans la sémantique opérationnelle, les transitions où les variables sont modifiées se retrouvent en fonction de la valeur de la ligne courante. La valeur de la variable pc est donc utilisée pour indiquer les lignes concernées. Par exemple, si les lignes 6 et 8 contiennent une affectation ST x, la définition de la variable x est la suivante :

```
next (x) := switch (pc)
    {
      6 : a;
      8 : a;
      default : x;
    };
```

La variable x prend la valeur de l'accumulateur lors des affectations ; elle conserve sa valeur le reste du temps. Une telle déclaration peut être mise en relation avec la sémantique opérationnelle, de la manière suivante : pour chaque transition $(V, a, i) \rightarrow (V', a', i')$, on a :

- $V'(\mathtt{x}) = a$ si $i = 6$ ou si $i = 8$;
- $V'(\mathtt{x}) = V(\mathtt{x})$ sinon.

6.2.4. Organisation modulaire

Le fichier de la spécification complète comporte un unique module, dans lequel sont définis les variables, l'accumulateur et la variable pc.

6.3. Sémantique directe ST

6.3.1. Principes

Le codage de la sémantique directe ST est relativement similaire au codage de la sémantique directe IL. L'accumulateur et la ligne courante sont ici remplacés par une variable entière codant la position courante, qui a pour rôle de cadencer l'exécution du programme. L'évolution des variables suit un schéma similaire à celui présenté précédemment.

6.3.2. Contrôle : continuation et cycle

La sémantique opérationnelle définit entièrement le comportement du paramètre indiquant la position. Nous savons, de plus, qu'il existe un nombre limité de positions en fonction du programme, et, qui plus est, ce nombre est linéaire en la taille du programme. Il reste à coder cette position par un entier, en tenant compte du fait que les états dont

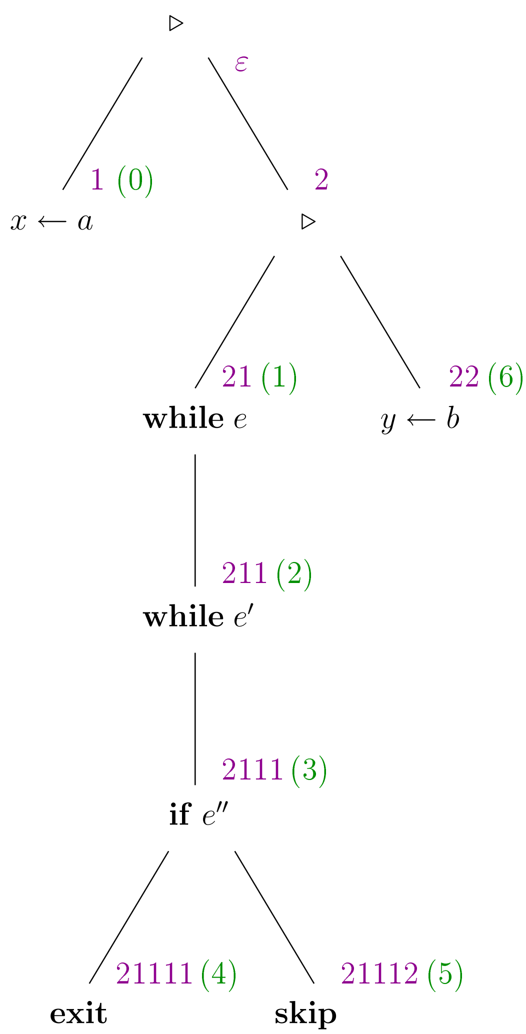

FIG. 6.1.: Numérotation des positions.

la position correspond à un nœud étiqueté par $\triangleright$ ne sont pas atteignables (comme décrit par la sémantique, on exécute directement le sous-terme gauche). Les états correspondant à l'opérateur $\triangleright$ sont donc exclus du codage des positions.

La méthode la plus simple consiste à numéroter les nœuds de l'arbre par un parcours préfixe en profondeur[1]. La figure 6.1 indique, entre parenthèses, la numérotation de l'arbre sur l'exemple de la section 4.2.4.6 page 61. Lors de la numérotation, on maintient une table de correspondance entre positions et numérotation, comme le montre le tableau 6.1. Si le nombre de numéros affectés est n, on numérote \$ par $n + 1$.

Du tableau, et de la liste des transitions déduite de la sémantique (voir le tableau page 61, et pour un exemple la section 4.2.4.6), nous déduisons l'évolution de la variable

[1] On notera que cela revient tout simplement à numéroter, de gauche à droite, pendant la lecture, le terme représentant le programme.

Position	Numéro
ε	n/a
1	0
2	n/a
21	1
22	6
211	2
2111	3
21111	4
21112	5
\$	7

TAB. 6.1.: Correspondance entre positions et numérotation dans notre exemple.

déterminant la position dans le programme (que nous appellerons `pos`). La section SMV définissant l'évolution de la position se définit comme suit : pour chaque liaison entre deux continuations, correspondant à une transition de la sémantique opérationnelle, on écrit une ligne d'affectation portant sur la variable `pos`. Cette ligne comprend :

- un test du numéro correspondant à la position de l'état de départ (la source de la transition) ;
- éventuellement, un test permettant d'identifier la condition de franchissement ;
- le numéro correspondant à la position de l'état d'arrivée (la cible de la transition).

La section correspondant à notre exemple est présentée à la figure 6.2.

6.3.3. Variables

De même que pour la sémantique IL, les variables du modèle sont représentées par des variables déclarées sous SMV, qui évoluent de la même manière. On retrouve donc la différence de comportement entre les variables d'entrée et les variables de sortie.

Les variables d'entrée évoluent de façon non déterministe lors de la phase d'acquisition des entrées. Cette phase commence juste après la fin de cycle, donc au numéro correspondant au symbole \$. Pour toute autre position, la variable conserve sa valeur. Si, par exemple, on considère la variable d'entrée `y`, et si le numéro correspondant à \$ est 7 (comme dans l'exemple précédent), on définit :

```
next(y) := case
  {
    pos = 7 : {0,1};
    default : y;
  };
```

De même que dans le codage du IL, les variables de sortie sont modifiées pour chaque position dans le programme où figure une affectation. Elles conservent leur valeur dans les autres cas.

```
next(pos) := case
  {
    pos = 0 : 1;
    (pos = 1) & e : 2;
    (pos = 1) & ¬e : 6;
    (pos = 2) & e′ : 3;
    (pos = 2) & ¬e′ : 1;
    (pos = 3) & e″ : 4;
    (pos = 3) & ¬e″ : 5;
    pos = 4 : 1;
    pos = 5 : 2;
    pos = 6 : 7;
    pos = 7 : 0;
  };
```

FIG. 6.2.: Pseudo-code SMV illustrant l'évolution de la variable `pos`.

6.3.4. Organisation modulaire

De même que dans le codage du IL, la spécification du programme se compose d'un unique module, les définitions des variables et de la position, ainsi que les propriétés à vérifier.

6.4. Sémantique modulaire ST

6.4.1. Principes

6.4.1.1. Modularité

Le codage des éléments du langage ST exploite le fait de pouvoir décrire un système en SMV de façon modulaire (voir la section 5.4.2.6 page 112). Nous coderons en effet par un module chaque automate de la sémantique opérationnelle modulaire ST, présentée à la section 4.3 page 65. Ces modules sont ensuite instanciés, pour composer le modèle du programme étudié. L'organisation générale de la spécification SMV, comprenant en particulier l'instanciation des modules, est décrite à la section 6.4.6 page 135.

6.4.1.2. Expressions et énoncés

De même que les automates ont une interface unifiée, les modules servant à les coder comportent de nombreux points communs. En particulier, on distingue, comme pour la sémantique opérationnelle, les automates d'expressions des automates d'énoncés.

6.4.1.3. Codage de la synchronisation : les signaux

Les automates de la sémantique opérationnelle utilisent des messages, ayant pour but de définir les modalités d'appel et de réponse entre automates. Pour assurer ces communications, le codage en SMV utilise des variables. Les types de ces variables sont les suivants :

```
typedef Appel {1, N};
typedef Signal {0, 1, N};
typedef Continuation {1, Exit, Return, N};
```

Les types de données sont utilisés pour spécifier les signaux pouvant être portés par chaque variable servant d'interface :
- `Appel` est le type des variables utilisées comme signaux d'entrée des modules (correspondant au message $?i$ des automates). La valeur `1` signifie un appel (lancer l'exécution ou l'évaluation) et la valeur `N` est un signal neutre (ne rien faire) ;
- `Signal` est le type des variables utilisées comme retour dans les expressions. Les valeurs `1` et `0` signifient une fin d'évaluation avec réponse vraie ou fausse (correspondant aux messages $!o = \texttt{TRUE}$ et $!o = \texttt{FALSE}$ des automates). La valeur `N` est un signal neutre (pas de réponse : le module n'est pas en phase de fin de calcul) ;
- `Continuation` est le type des variables utilisées comme retour dans les énoncés. Les valeurs `1` (terminaison normale), `Exit` et `Return` correspondent respectivement aux signaux $!o$, $!exit$ et $!return$ des automates. Comme pour les expressions, le signal neutre `N` est utilisé pour signifier l'absence de message.

On code donc « à la main » le mécanisme de synchronisation, en définissant explicitement les phases d'attente des automates (leurs variables de sortie n'évoluent pas) d'après les entrées.

6.4.2. Codage des expressions

6.4.2.1. Forme générale des modules

Les modules d'expression requièrent au moins deux arguments, `i` et `o`. L'argument `i` est de type `Appel` et est une entrée pour le module, l'argument `o` est de type `Signal` et est une sortie pour le module. D'autres arguments peuvent s'ajouter, en particulier les variables d'interface d'autres modules (si celui-ci doit communiquer avec d'autres), ou une variable booléenne dans le cas d'une variable vue comme une expression. Chaque module doit définir le comportement des variables définies pour lui comme des sorties, et en particulier la variable `o`, qui doit être conforme à la description suivante :
- prendre la valeur `1` si l'expression se termine avec la valeur vraie, au moment précis de la fin de l'évaluation ;
- prendre la valeur `0` si l'expression se termine avec la valeur fausse, au moment précis de la fin de l'évaluation ;
- prendre la valeur neutre `N` dans tous les autres cas (c'est-à-dire à tout moment autre que la fin de l'évaluation).

6.4.2.2. Modules représentant les expressions

Variable vue comme une expression Le codage en SMV de l'automate correspondant à une variable vue comme une expression est le suivant :

```
module expr_variable (i, var, o)
  {
    input i : Appel;
    output o : Signal;
    init (o) := N;
    next (o) := case
                  {
                    i = 1 & var = 1 : 1;
                    i = 1 & var = 0 : 0;
                    i = N : N;
                  };
  }
```

Le module prend la variable en argument. La valeur de `o` est mise à jour à la transition qui suit la mise à `1` de la variable `i`. La variable `o` prend alors la valeur de `var`. Ce module permet d'intégrer une variable dans une expression en y apportant une interface conforme à l'interface (`i`, `o`) des modules d'expressions.

Expression vraie Le codage en SMV de l'automate correspondant à l'expression vraie est le suivant :

```
module expr_vrai (i, o)
  {
    input i : Appel;
    output o : Signal;
    init (o) := N;
    next (o) := case
                  {
                    i = N : N;
                    i = 1 : 1;
                  };
  }
```

Le module ne nécessite aucun autre argument. La variable `o` prend la valeur `1` à la transition qui suit la mise à `1` de la variable `i`.

Expression fausse Le codage en SMV de l'automate correspondant à l'expression fausse est le suivant :

```
module expr_faux (i, o)
  {
```

```
    input i : Appel;
    output o : Signal;
    init (o) := N;
    next (o) := case
                  {
                    i = N : N;
                    i = 1 : 0;
                  };
  }
```

De même que pour l'expression vraie, le module ne nécessite aucun autre argument. La variable o prend la valeur `0` à la transition qui suit la mise à `1` de la variable `i`.

Négation Le codage en SMV de l'automate correspondant à la négation est le suivant :

```
module expr_negation (i, ie, oe, o)
  {
    input i : Appel;
    output ie : Appel;
    input oe : Signal;
    output o : Signal;
    init (ie) := N;
    init (o) := N;
    next (ie) := i;
    next (o) := case
                  {
                    oe = N : N;
                    oe = 1 : 0;
                    oe = 0 : 1;
                  };
  }
```

Le module prend en argument l'interface (`ie`, `oe`) d'un autre module d'expression. La variable `ie` est déclarée comme sortie, car elle correspond à un envoi de message : c'est le module de la négation qui va déclencher l'évaluation de la sous-expression en faisant passer `ie` à `1`. La variable `ie` est d'ailleurs systématiquement mise à jour avec la valeur de `i` à l'étape précédente. La variable `oe` est pour le module une entrée, elle correspond à la réception du résultat de l'évaluation de la sous-expression. La variable `o` est mise à jour avec la négation de la valeur de `oe` à la transition précédente (ou `N` si `N` était cette valeur).

Conjonction Le codage en SMV de l'automate correspondant à la conjonction est le suivant :

```
module expr_conjonction (i, i1, o1, i2, o2, o)
  {
```

```
    input i : Appel;
    output i1, i2 : Appel;
    input o1, o2 : Signal;
    output o : Signal;
    init (i1) := N;
    init (i2) := N;
    init (o) := N;
    next (i1) := i;
    next (i2) := case
                   {
                     o1 = 0 : N;
                     o1 = 1 : 1;
                     default : N;
                   };
    next (o) := case
                  {
                    o1 = 0 : 0;
                    default : o2;
                  };
  }
```

Ici, le module prend en argument les interfaces de deux sous-expressions. Les `o1` et `o2` sont des entrées pour le module, et les `i1` et `i2` sont des sorties. La variable `i1` est mise à jour avec la valeur de `i` à l'étape précédente. La variable `i2` est mise à jour avec la valeur de `o1`, seulement si celle-ci vaut `1`. Cela correspond au mode d'évaluation défini par l'automate de la conjonction : si la première sous-expression est fausse, on n'évalue pas la deuxième. La valeur de `o` est donc mise à jour avec la valeur de `o1` à l'étape précédente si celle-ci est fausse (cas où `o2` n'a pas été évaluée), et avec la valeur de `o2` à l'étape précédente sinon.

Disjonction Le codage en SMV de l'automate correspondant à la disjonction est le suivant :

```
module expr_disjonction (i, i1, o1, i2, o2, o)
  {
    input i : Appel;
    output i1, i2 : Appel;
    input o1, o2 : Signal;
    output o : Signal;
    init (i1) := N;
    init (i2) := N;
    init (o) := N;
    next (i1) := i;
    next (i2) := case
                   {
```

```
                    o1 = 1 : N;
                    o1 = 0 : 1;
                    default : N;
                   };
     next (o) := case
                  {
                    o1 = 1 : 1;
                    default : o2;
                  };
   }
```

De même que pour la conjonction, le module prend en argument les interfaces de deux sous-expressions. L'évolution des variables de sortie suit le même principe excepté le fait que c'est lorsque la variable `o1` vaut `0` que l'évaluation de la deuxième sous-expression a lieu. La variable `o` est alors mise à jour avec la valeur de `o1` à l'étape précédente si celle-ci est vraie (cas où `o2` n'a pas été évaluée), et avec la valeur de `o2` à l'étape précédente sinon.

6.4.3. Codage des énoncés

6.4.3.1. Forme générale des modules

Les modules d'énoncé requièrent au moins deux arguments, `i` et `o`. L'argument `i` est de type `Appel` et est une entrée pour le module, l'argument `o` est de type `Continuation` et est une sortie pour le module. D'autres arguments peuvent s'ajouter, en particulier les variables d'interface d'autres modules (si celui-ci doit communiquer avec d'autres). Chaque module contient une spécification pour chaque variable déclarée en `output`. C'est en particulier le cas de la variable `o`, qui doit avoir le comportement suivant :

– prendre la valeur `1` si l'énoncé se termine par une terminaison normale ;
– prendre la valeur `Exit` si l'énoncé se termine par une terminaison par `EXIT` ;
– prendre la valeur `Return` si l'énoncé se termine par une terminaison par `RETURN` ;
– prendre la valeur neutre `N` dans tous les autres cas.

6.4.3.2. Modules représentant les énoncés

Affectation Le codage en SMV de l'automate correspondant à l'affectation est le suivant :

```
module enonce_affectation (i, ie, oe, o)
  {
    input i : Appel;
    output ie : Appel;
    input oe : Signal;
    output o : Continuation;
    init (ie) := N;
    init (o) := N;
```

```
        next (ie) := i;
        next (o) := case
                      {
                         oe = 1 : 1;
                         oe = 0 : 1;
                         default : N;
                      };
      }
```

Selon la règle de l'affectation unique sous SMV, il n'est pas possible d'affecter localement une variable définie dans un autre module. En effet, chaque variable ne peut être affectée qu'une seule fois au plus, et il n'est pas autorisé de spécifier deux affectations concurrentes de la même variable. Pour cela, il est préférable que chaque variable soit affectée dans le module où elle est déclarée. C'est pourquoi la variable elle-même n'apparaît pas parmi les arguments du module.

Les arguments `ie` et `oe` concernent l'expression à évaluer pour déterminer la valeur à affecter. L'énoncé en lui-même ne fait que provoquer l'évaluation de l'expression en mettant à jour la variable `ie` avec la valeur de la variable `i` à l'étape précédente, puis affecter à `o` la valeur de terminaison normale lorsque `oe` porte une valeur, quelle qu'elle soit. L'affectation de la variable en elle-même doit être pratiquée dans le module où la variable est déclarée, en scrutant la valeur de `oe`, le résultat de l'expression (voir la section 6.4.4.2 page 134).

Énoncé vide Le codage en SMV de l'automate correspondant à l'énoncé vide est le suivant :

```
    module enonce_tau (i, o)
      {
        input i : Appel;
        output o : Continuation;
        init (o) := N;
        next (o) := case
                      {
                        i = 1 : 1;
                        i = N : N;
                      };
      }
```

Énoncé EXIT Le codage en SMV de l'automate correspondant à l'énoncé `EXIT` est le suivant :

```
    module enonce_exit (i, o)
      {
        input i : Appel;
        output o : Continuation;
```

```
    init (o) := N;
    next (o) := case
                  {
                    i = 1 : Exit;
                    i = N : N;
                  };
  }
```

Énoncé RETURN Le codage en SMV de l'automate correspondant à l'énoncé `RETURN` est le suivant :

```
module enonce_return (i, o)
  {
    input i : Appel;
    output o : Continuation;
    init (o) := N;
    next (o) := case
                  {
                    i = 1 : Return;
                    i = N : N;
                  };
  }
```

Séquence Le codage en SMV de l'automate correspondant à la séquence est le suivant :

```
module enonce_sequence (i, i1, o1, i2, o2, o)
  {
    input i : Appel;
    output i1, i2 : Appel;
    input o1, o2 : Continuation;
    output o : Continuation;
    init (i1) := N;
    init (i2) := N;
    init (o) := N;
    next (i1) := i;
    next (i2) := case
                   {
                     o1 = 1 : 1;
                     o1 = Exit : N;
                     o1 = Return : N;
                     default : N;
                   };
    next (o) := case
                  {
```

```
                        o1 = Exit | o2 = Exit : Exit;
                        o1 = Return | o2 = Return : Return;
                        o2 = 1 : 1;
                        default : N;
                      };
      }
```

L'énoncé prend en argument les variables d'interface de deux sous-énoncés. L'exécution du deuxième sous-énoncé n'est provoquée que dans le cas où l'exécution du premier énoncé s'est terminée par un signal de terminaison normale. Le signal de sortie `o` est mis à jour avec le signal `Exit` ou `Return` du premier ou du deuxième sous-énoncé (si un de ces deux signaux a été envoyé c'est forcément par un seul des deux), ou avec le signal de terminaison normale si le deuxième sous-énoncé s'est terminé par ce signal (si le deuxième a été exécuté, c'est forcément que le premier a envoyé ce signal lui aussi). Dans tous les autres cas, `o` est affecté du signal neutre.

Sélection Le codage en SMV de l'automate correspondant à la sélection est le suivant :

```
module enonce_selection (i, ie, oe, i1, o1, i2, o2, o)
  {
    input i : Appel;
    output i1, i2, ie : Appel;
    input oe : Signal;
    input o1, o2 : Continuation;
    output o : Continuation;
    init (ie) := N;
    init (i1) := N;
    init (i2) := N;
    init (o) := N;
    next (ie) := i;
    next (i1) := case
                   {
                     oe = 1 : 1;
                     default : N;
                   };
    next (i2) := case
                   {
                     oe = 0 : 1;
                     default : N;
                   };
    next (o) := case
                  {
                    o1 = Exit | o2 = Exit : Exit;
                    o1 = Return | o2 = Return : Return;
```

```
                o1 = 1 | o2 = 1 : 1;
                default : N;
              };
    }
```

Le module prend en argument les signaux correspondant à une expression et deux énoncés. L'expression est évaluée, puis l'un des deux énoncés est exécuté. Son signal de retour est alors affecté à la variable `o`.

Itération Le codage en SMV de l'automate correspondant à l'itération est le suivant :

```
module enonce_iteration (i, ie, oe, is, os, o)
  {
    input i : Appel;
    output ie, is : Appel;
    input oe : Signal;
    input os : Continuation;
    output o : Continuation;
    init (ie) := N;
    init (is) := N;
    init (o) := N;
    next (ie) := case
               {
                 i = 1 : 1;
                 os = 1 : 1;
                 default : N;
               };
    next (is) := case
               {
                 oe = 1 : 1;
                 default : N;
               };
    next (o) := case
              {
                oe = 0 : 1;
                os = Exit : 1;
                os = Return : Return;
                default : N;
              };
  }
```

Le module prend en argument les signaux correspondant à une expression et un énoncé. L'expression est évaluée à l'exécution de l'énoncé (`i = 1`) et à chaque fin d'exécution du sous-énoncé (`os = 1`). Le sous-énoncé est exécuté à chaque résultat vrai de l'expression (`oe = 1`). Le résultat de l'énoncé est une terminaison normale si l'expression donne un résultat faux (`oe = 0`) ou si le sous-énoncé s'est terminé par un exit (`os = Exit`). L'énoncé se termine par un `Return` si le sous-énoncé s'est terminé par un `Return` (`os = Return`).

6.4.4. Variables

6.4.4.1. Entrées

Comme dans la sémantique directe, les variables d'entrée évoluent juste après la fin de cycle. La fin de cycle est indiquée par le signal de sortie du module représentant l'énoncé principal , que nous appellerons ici `fin` (voir la section 6.4.5). La fin de cycle intervient lorsque la valeur de la variable `fin` n'est pas `N` (une autre valeur indique en effet que le module principal est arrivé en fin d'exécution).

```
init (ann_req) := 0;
next (ann_req) := case
  {
    fin ~= N : {0, 1};
    default  : ann_req;
  };
```

6.4.4.2. Sorties et internes

Les variables évoluent en fonction des énoncés d'affectation qui les concernent. Par exemple, si `affectation_var` est une instance du module `enonce_affectation` et `var` la variable à affecter, le code suivant permet d'affecter la variable `var` en accord avec les règles de l'affectation :

```
var : case
        {
          affectation_var.oe = 1 : 1;
          affectation_var.oe = 0 : 0;
          ⋮
          default : N;
        };
```

Les points de suspension représentent d'éventuelles autres affectations pouvant dépendre d'autres énoncés.

6.4.5. Contrôle

6.4.5.1. Continuation

Le séquencement des instructions du programme est naturellement assuré par les appels entre modules. Il suffit d'instancier les modules concernés en mettant leurs signaux d'entrée et de sortie en correspondance, comme décrit un peu plus loin (section 6.4.6 page ci-contre).

6.4.5.2. Cycle

En raison des emboîtements modulaires sur lesquels le codage est construit, le contrôle du programme entier est naturellement effectué par les signaux d'entrée et de sortie du module représentant l'énoncé principal (lorsqu'on représente le programme sous forme arborescente, l'énoncé principal est celui qui occupe la racine). Supposons par exemple que l'énoncé principal soit une séquence (énoncé $\triangleright$) et que ses signaux d'entrée et de sortie soient respectivement `seq1_i` et `seq1_o`. Pour plus de clarté, on définit éventuellement des symboles `debut` et `fin` leur correspondant :

```
#define debut seq1_i
#define fin seq1_o
```

Le comportement du signal `fin` est défini par le module correspondant (ici, le module de la séquence). La fin de cycle correspond donc au moment où ce signal est activé, c'est-à-dire lorsqu'il porte une valeur non neutre (`fin ~= N`).

Le signal `debut` a ainsi pour charge de déclencher le nouveau cycle, en fonction de la réponse du signal `fin` :

```
init (debut) := 1;
next (debut) := case
  {
    fin = 1      : 1;
    fin = Return : 1;
    default      : N;
  };
```

Notons que, puisque le programme est supposé bien formé, il ne comporte pas d'énoncé `EXIT` situé en dehors d'un `WHILE`. Par conséquent, le signal `Exit` ne peut pas remonter jusqu'à l'énoncé principal, et il n'est donc pas nécessaire de le prendre en compte (on pourrait à la rigueur, si un tel contrôle n'a pas été effectué en amont, ajouter ici une redirection vers un état d'erreur).

6.4.6. Organisation modulaire

Le codage d'un programme ST se compose de deux parties : une partie constante, composée du codage des éléments de programme (défini ci-dessus) et une partie variable, représentant le programme lui-même. Les éléments de programme sont donnés sous la forme de modules ; un programme ST est lui-même codé par un module, dans lequel sont instanciés des modules correspondant aux éléments présents dans le programme. Les liaisons entre les instances sont alors définies en rapport avec la structure du programme. Par exemple, considérons la partie de programme ST suivante :

```
a := NOT b;
```

Ce programme est composé de quatre éléments : deux variables, une négation, une affectation. Pour coder ce programme, nous définissons un module destiné à le représenter.

```
module main ()
  {
```

Nous ajoutons une déclaration des variables utilisées dans le programme :

```
b : boolean;
a : boolean;
```

Nous aurons besoin d'instancier les modules suivants :
- un module de variable vue comme expression pour la variable `b`, que nous appellerons `var_b` ;
- un module de négation, que nous appellerons `neg` ;
- un module d'affectation, que nous appellerons `aff`.

Nous déclarons les variables d'interface utilisées par ces modules :

```
var_b_i, neg_i, aff_i : Appel;
var_b_o, neg_o : Signal;
aff_o : Continuation;
```

Nous instancions les modules, en rendant explicites leurs liaisons, par l'usage des variables :

```
var_b : expression_variable (var_b_i, b, var_b_o);
neg : expression_negation (neg_i, var_b_i, var_b_o, neg_o);
aff : enonce_affectation (aff_i, neg_i, neg_o, aff_o);
```

Nous identifions les signaux de début de de fin, d'après le nom des signaux de l'énoncé principal (ici `aff`) :

```
#define debut aff_i
#define fin aff_o
```

Nous ajoutons, pour terminer, une spécification de nos deux variables. La variable `a` est une variable de sortie, affectée par l'énoncé `aff` :

```
init (a) := 0;
next (a) := case
              {
                aff.oe = 1 : 1;
                aff.oe = 0 : 0;
                default : a;
              };
```

La variable b est une variable d'entrée :

```
init (b) := 0;
next (b) := case
  {
    fin ~= N : {0, 1};
    default  : b;
  };
```

6.5. Conclusion

Ce chapitre a présenté un codage de chacune des trois sémantiques opérationnelles définies au chapitre 4. Des exemples concrets de chacun des trois codages sont présentés au chapitre suivant. Les trois méthodes ont en commun de définir un comportement pour toutes les variables, et en particulier celui des variables d'entrée (car l'évolution des entrées est dépendante du cycle de l'automate). Puisque le comportement de toutes les variables, y compris d'entrée, est défini, le problème lié au changement de modèle de Cadence SMV (voir la section 5.4.3.4 page 116) ne se pose pas : toutes les formules CTL peuvent être vérifiées sans surprise.

Chacune de ces trois méthodes est déduite de la sémantique dont elles est inspirée, et ce de manière assez directe. Si, comme on l'imagine, le codage de la sémantique directe est performant, celui de la sémantique modulaire, en raison du grand nombre de variables qu'il engendre, offre des performances plus limitées. Il est cependant possible d'effectuer la vérification de systèmes de taille raisonnable, comme par exemple celui de la section 7.3.3 page 153. Le troisième codage peut faire l'objet d'un travail d'optimisation, qui permettrait, tout en conservant le lien avec la sémantique d'origine (qui serait pour lors moins évident, et nécessiterait d'être formellement prouvé). Lors de ce travail, nous nous sommes bornés à indiquer le codage le plus direct possible, afin de garder évident le rapport avec la sémantique opérationnelle.

7. Exemples

7.1. Introduction

Nous illustrons la méthode présentée dans les chapitres précédents au moyen de deux exemples complets : l'exemple du mixeur, directement extrait de la norme, et l'exemple du chariot, qui a fait l'objet d'une publication [CDP+00]. Après sa présentation, nous donnons pour chaque exemple le ou les programmes qui les implémentent, les codages associés, et une sélection de propriétés que nous faisons vérifier par SMV.

7.2. Mixeur

7.2.1. Présentation

7.2.1.1. Description

L'exemple du mixeur, illustré par la figure 7.1, est un exemple réel présenté par la norme (annexe F5). Il met en œuvre un système de mixage permettant de mélanger des briques d'un produit solide avec deux liquides. Il est composé de trois sous-systèmes :
- une unité de remplissage des liquides ;
- une unité de dépôt des briques ;
- une unité de mixage.

Cahier des charges Le fonctionnement du système se décrit comme suit :

1. à l'état initial, le système est à l'arrêt, le bol mixeur est en position haute ; un bouton `ST` déclenche la mise en route du cycle de mixage ;
2. les produits à mélanger sont déposés dans le mixeur, de la façon suivante :
   a) le liquide A est versé dans la cuve C jusqu'à obtention du poids a requis, puis le produit B est versé à son tour dans la cuve C jusqu'à obtention du poids $a + b$ requis. Le mélange $A + B$ est ensuite versé dans le mixeur,
   b) exactement deux briques sont versées dans le mixeur. Le temps de chute d'une brique étant supérieur à la période de cycle de l'automate programmable industriel, la présence de chaque brique est constatée plusieurs fois devant le capteur d.

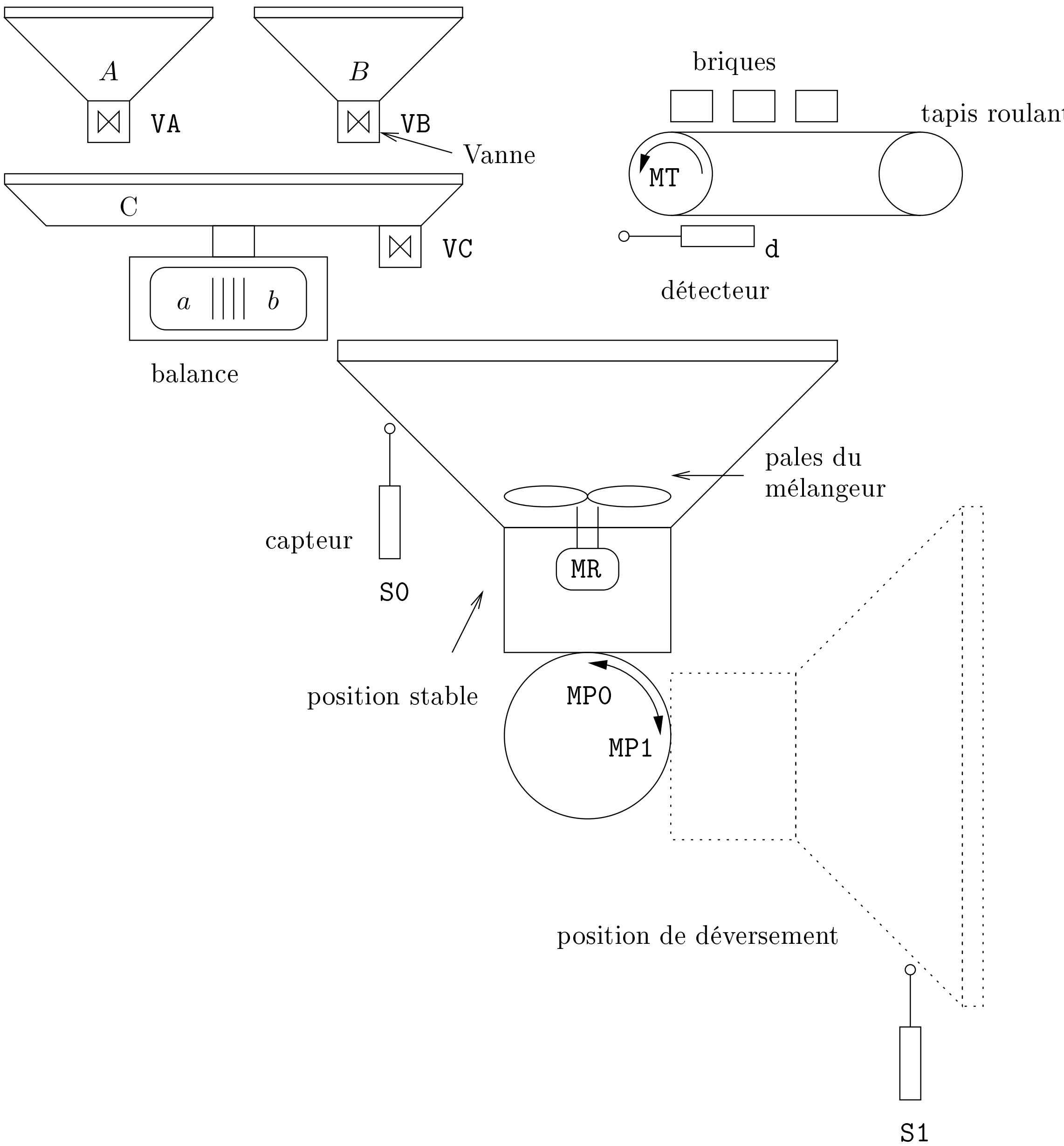

FIG. 7.1.: Représentation du mixeur.

C'est l'alternance de détection/non-détection qui permet de distinguer la chute de deux briques différentes ;

3. les briques et le liquide versés, le mixeur `MR` se met en marche, pendant que l'ensemble de la cuve bascule jusqu'au butoir `S1`. La cuve reste basculée jusqu'à l'écoulement d'un délai t_1, pendant lequel le mixeur continue à tourner. Puis l'ensemble de la cuve revient en place, jusqu'à atteindre le butoir `S0`.

Les parties 2 et 3 s'effectuent en alternance, et en exclusion mutuelle : il n'est, par exemple, pas possible de verser le liquide A dans sa cuve pendant que le mixeur tourne. Par contre, les parties 2a et 2b peuvent s'effectuer en parallèle.

Prise en compte des temporisations Notre modèle ne comprend pas de prise en compte des temporisations, mais cela n'est pas nécessaire pour traiter l'exemple. En effet, « le délai t_1 est écoulé » peut être vu, de façon logique, comme une condition booléenne. Cette condition sera considérée comme une entrée booléenne (`MIX_TEMPS`).

7.2.1.2. Signaux

Tous les signaux d'entrée et de sortie sont booléens. Ils sont décrits ci-après :

Signaux d'entrée

`ST` bouton de démarrage (autorise de commencer le cycle de mixage) ;

`d` détecteur de transit (détecte la présence d'une brique dans son champ de vision) ;

`S0` indicateur de la limite haute de la bascule ;

`S1` indicateur de la limite basse de la bascule ;

`BALANCE_VIDE` si à `TRUE`, le poids mesuré sur la balance n'excède pas la tare ;

`POIDS_A` si à `TRUE`, le poids mesuré sur la balance dépasse le poids de la tare, plus celui de A ;

`POIDS_AB` si à `TRUE`, le poids mesuré sur la balance dépasse le poids de la tare, plus celui de A et celui de B ;

`MIX_TEMPS` si à `TRUE`, le temps limite de mixage est atteint.

Signaux de sortie

`VA` commande de la valve A (`TRUE` : ouverte, `FALSE` : fermée) ;

`VB` commande de la valve B (id.) ;

`VC` commande de la valve C (id.) ;

`MT` moteur de la courroie de distribution des briques ;

`MR` moteur de rotation du mixeur ;

`MP0` commande de levage de la bascule ;

`MP1` commande d'abaissement de la bascule.

7.2.2. Mise en œuvre

7.2.2.1. Spécification SFC

Dans le domaine de l'automatique, une spécification en langage SFC accompagne fréquemment un cahier des charges textuel. Elle le complète en décrivant précisément les différentes phases et les interactions avec l'extérieur. Le langage SFC est un langage de spécification graphique pour automates programmables industriels, apparenté au grafcet ; c'est l'un des cinq langages définis par la norme IEC 61131-3. Nous ne décrirons pas ici l'interprétation générale du langage SFC, mais nous commenterons la spécification présentée dans le cas précis de notre exemple. Cela, et le caractère simple et relativement intuitif du langage SFC, apporteront au lecteur les éléments nécessaires pour bien comprendre la spécification.

La spécification du mixeur en SFC est donnée par la figure 7.2. Elle distingue clairement trois sous-parties du système, où nous retrouvons les parties 2a, 2b et 3 du cahier des charges (section 7.2.1.1).

Les blocs reliés entre eux et encadrés du diagramme sont les *étapes*. Les liaisons entre les étapes sont les *transitions*. L'étiquette indiquée de la transition est la *condition de franchissabilité* : c'est une formule booléenne qui détermine le passage de l'étape en amont à l'étape en aval. Chaque étape peut être directement reliée à un bloc : c'est l'*action associée* à l'étape. L'étape `START`, encadrée d'un double trait, est l'étape initiale.

La première condition à remplir pour quitter l'étape `START` est une conjonction de trois conditions :
- l'utilisateur autorise le début du cycle de mixage (signal `ST`) ;
- la bascule du mixeur est en position haute (signal `S0`) ;
- la balance des cuves à liquide est vide (signal `BALANCE_VIDE`).

Lorsque le cycle de mixage peut commencer, il exécute chacune des deux sous-parties 2a et 2b en parallèle. Lorsque ces deux parties sont terminées (les deux étapes `FILL` et `DROP_2` sont simultanément actives) *et* que la condition de franchissement est remplie (`POIDS_AB & NOT d`, c'est-à-dire que la cuve des liquides mélangés est vidée et la deuxième brique a fini de tomber), alors la partie 3 est activée. Lorsqu'elle est terminée, le système revient dans l'état initial (`START`).

Le déroulement des actions du système pour chaque sous-partie est le suivant :

1. le mélange des liquides et la chute des briques se déroulent simultanément, à savoir :

   a) Dans l'étape `WEIGH_A`, le système ouvre la vanne du liquide A (`N VA` – la commande `N` déclenche la levée du signal placé en argument ; le signal est stoppé aussitôt que « le bouton est lâché », c'est-à-dire que l'étape contenant le `N` n'est plus active). La vanne reste ouverte jusqu'à atteindre le poids requis pour le liquide A, puis l'étape `WEIGH_B` est activée (et donc `WEIGH_A` désactivée). La vanne du liquide B est alors ouverte jusqu'à obtention du poids requis pour $A + B$. Le système ouvre alors la vanne de la grande cuve, puis se place en attente, dans l'étape `FILL`.

```
+--------->-----------+
|                     |
|                +====+====+   +---+------+
|                || START ||---| N | DONE |
|                +====+====+   +---+------+
|                     |
|                     + ST & S0 & BALANCE_VIDE
|    ===+=============+===============+=======
|       |                             |
|  +----+----+   +---+----+      +----+---+   +---+----+
|  | WEIGH_A |---| N | VA |      | BRICK1 |---| S | MT |
|  +----+----+   +---+----+      +----+---+   +---+----+
|       |                             |
|       +POIDS_A                      + d
|       |                             |
|  +----+----+   +---+----+      +----+---+
|  | WEIGH_B |---| N | VB |      | DROP_1 |
|  +----+----+   +---+----+      +----+---+
|       |                             |
|       + POIDS_AB                    + NOT d
|       |                             |
|  +----+----+   +---+----+      +----+---+
|  |  FILL   |---| N | VC |      | BRICK2 |
|  +----+----+   +---+----+      +----+---+
|       |                             + d
|       |                        +----+---+   +---+----+
|       |                        | DROP_2 |---| R | MT |
|       |                        +----+---+   +---+----+
|       |                             |
|   ====+=============+===============+=====
|                     |
|                     + BALANCE_VIDE & NOT d
|                     |
|                  +--+--+   +---+----+
|                  | MIX |---| S | MR |
|                  +--+--+   +---+----+
|                     |
|                     + MIX_TEMPS
|                     |
|                  +--+--+   +---+-----+
|                  | TIP |---| N | MP1 |
|                  +--+--+   +---+-----+
|                     |
|                     + S1
|                     |
|                 +---+---+   +---+-----+
|                 | RAISE |---| R | MR  |
|                 +---+---+   +---+-----+
|                     + S0    | N | MP0 |
|                     |       +---+-----+
+----------<----------+
```

Fig. 7.2.: Spécification du mixeur en SFC.

   b) Dans l'étape `BRICK1`, le système met en route le moteur du tapis distributeur des briques (la commande S déclenche la levée du signal placé en argument ; le signal n'est stoppé que par une commande R avec le même argument). Dès que le signal `d` est reçu (une brique est entrée dans le champ de vision du capteur), le système active l'étape `DROP_1` et attend la fin de la chute de la première brique. Lorsque plus rien n'entre dans le champ de vision du capteur, le système attend la chute de la deuxième brique. Lorsqu'elle commence sa chute, le moteur du tapis roulant est stoppé, puis le système se met en attente, dans l'étape `DROP_2`.

2. Lorsque la cuve de mixage est pleine, le moteur du mixeur est mis en route pendant un délai t_1, puis, à l'étape `TIP`, la cuve bascule vers le bas, en continuant de mixer. Dès la réception du signal `S1`, le mixeur cesse de tourner, puis la cuve remonte. À la réception du signal `S0` (cuve en position maximale haute), la cuve stoppe, et le système revient à l'état initial.

7.2.2.2. Programme

Nous disposons du programme suivant, écrit en IL, qui implémente la commande du mixeur.

```
VAR_INPUT
        ST : bool; (* Start ou stop *)
        BALANCE_VIDE : bool; (* BCD_TO_INT(WC) <= z *)
        POIDS_A : bool; (* BCD_TO_INT(WC) >= WA + Z *)
        POIDS_AB : bool; (* BCD_TO_INT(WC) >= WA + WB + Z *)
        MIX_TEMPS :  bool; (* MIX.T >= t1 *)
        d : bool;
        S0 : bool;
        S1 : bool;
END_VAR

VAR_OUTPUT
        DONE : bool := TRUE;
        VA : bool := FALSE;
        VB : bool := FALSE;
        VC : bool := FALSE;
        MT : bool := FALSE;
        MR : bool := FALSE;
        MP1 : bool := FALSE;
        MP0 : bool := FALSE;
END_VAR

VAR
        START : bool := TRUE;
        WEIGH_A : bool := FALSE;
        WEIGH_B : bool := FALSE;
        FILL : bool := FALSE;
        BRICK1 : bool := FALSE;
        DROP_1 : bool := FALSE;
```

```
        BRICK2 : bool := FALSE;
        DROP_2 : bool := FALSE;
        MIX : bool := FALSE;
        TIP : bool := FALSE;
        RAISE : bool := FALSE;
END_VAR

start:  LD      START
        JMPCN   weigh_a
        (* condition start *)

        LD      ST
        AND     S0
        AND     BALANCE_VIDE
        JMPCN   end

        R       DONE
        R       START
        S       WEIGH_A
        S       VA
        S       BRICK1
        S       MT

(* Partie concernant les poids *)
weigh_a: LD     WEIGH_A
        JMPCN   weigh_b

        LD      POIDS_A
        JMPCN   brick1

        R       VA
        R       WEIGH_A
        S       VB
        S       WEIGH_B

weigh_b: LD     WEIGH_B
        JMPCN   fill

        LD      POIDS_AB
        JMPCN   brick1

        R       VB
        R       WEIGH_B
        S       VC
        S       FILL

fill:   (* rien, voir drop_2 *)

(* Partie concernant les briques *)
brick1: LD      BRICK1
        JMPCN   drop_1
```

```
        LD      d
        JMPCN   end

        R       BRICK1
        S       DROP_1

drop_1: LD      DROP_1
        JMPCN   brick2

        LDN     d
        JMPCN   end

        R       DROP_1
        S       BRICK2

brick2: LD      BRICK2
        JMPCN   drop_2

        LD      d
        JMPCN   end

        R       BRICK2
        R       MT
        S       DROP_2

drop_2: LD      DROP_2
        JMPCN   mix

        AND     FILL
        ANDN    d
        AND     BALANCE_VIDE
        JMPCN   end

        R       VC
        R       FILL
        R       DROP_2
        S       MIX
        S       MR

(* Partie finale *)
mix:    LD      MIX
        JMPCN   tip

        LD      MIX_TEMPS
        JMPCN   end

        R       MIX
        S       MP1
        S       TIP
```

```
tip:    LD      TIP
        JMPCN   raise

        LD      S1
        JMPCN   end

        R       TIP
        R       MP1
        R       MR
        S       RAISE
        S       MP0

raise:  LD      RAISE
        JMPCN   end

        LD      S0
        JMPCN   end

        R       RAISE
        R       MP0
        S       DONE
        S       START

end:
```

De la sémantique opérationnelle et de la méthode de codage indiquée à la section 6.2 page 119, nous déduisons le code SMV donné à l'annexe D page 193.

7.2.3. Vérification de propriétés

7.2.3.1. Introduction

Le code IL donné à l'annexe D est issu du cahier des charges de la spécification SFC. La méthode de codage utilisée ne sera pas décrite ici[1] ; on s'attachera à vérifier aussi bien la validité de ce codage que celle du programme IL lui-même. Pour cela, nous évoquons trois types de propriétés :

- des propriétés sur les variables du système, qui contribuent à la validation du programme, sans prendre en compte la méthode par laquelle il a été obtenu ;
- des propriétés mettant en relation des variables du système et des éléments du SFC (noms d'étapes) : ces propriétés permettent de valider aussi bien le programme lui-même que la méthode de codage du SFC en IL, par laquelle le programme a été obtenu ;
- d'autres propriétés sur les étapes du SFC. Elles permettent de vérifier le comportement du SFC lui-même, en même temps que le codage de SFC en IL.

Nous présentons ci-après quelques exemples de propriétés vérifiées sur le système du mixeur.

[1] Le codage utilisé apparaît relativement clairement à la lecture du code.

7.2.3.2. Propriétés

Programme attentif Nous vérifions que le programme est attentif, par la formule LTL suivante (voir la section 5.2.5.2 page 100) :

$$\mathsf{G}\ \mathsf{F}\ fdc$$

Atteignabilité Nous vérifions dans un premier temps l'absence de code mort. Pour chaque ligne de programme i, on vérifie la formule CTL $\mathsf{AG}\ \mathsf{EF}\ (\mathtt{pc} = i)$ (voir la section 5.2.5.4 page 101).

Nous vérifions de manière analogue l'absence d'étapes mortes dans le SFC d'origine. Pour une étape E, cela se fait par la formule $\mathsf{AG}\ \mathsf{EF}\ E$. Cependant, pour les raisons invoquées à la section 5.2.5.5 page 102, nous souhaitons limiter l'observation aux seuls états de fin de cycle. Pour une étape E, cela se fait par la formule

$$\mathsf{AG}\ (fdc \Rightarrow \mathsf{EF}\ (E \wedge fdc))$$

Exclusions mutuelles On vérifie l'exclusion mutuelle, au moment de la fin de cycle, entre les variables a et b, au moyen de la formule LTL $\mathsf{G}\ (fdc \Rightarrow \neg(a \wedge b))$ (voir la section 5.2.5.5 page 102). Dans notre exemple, les variables en exclusion mutuelle sont les variables `VA` et `VB`, `VA` et `VC`, `VB` et `VC`, `MP0` et `MP1`, et enfin `MP0` et `MR`.

On peut étendre cette formule en vérifiant l'exclusion mutuelle entre la partie distribution et le mixage (parties haute et basse du schéma de la figure 7.1). La « partie haute » est représentée par les variables `VA`, `VB`, `VC` et `MT`. On dit que cette partie du système est en activité si et seulement si l'une de ces variables vaut `TRUE`. La « partie basse » est représentée par les variables `MR`, `MP0` et `MP1`. De même, on dit qu'elle est en activité si et seulement si l'une de ces variables vaut `TRUE`. L'exclusion mutuelle entre ces deux parties (au moment de la fin de cycle), mentionnée dans le cahier des charges, est représentée par la formule LTL

$$\mathsf{G}\ (fdc \Rightarrow \neg((\mathtt{VA} \vee \mathtt{VB} \vee \mathtt{VC} \vee \mathtt{MT}) \wedge (\mathtt{MR} \vee \mathtt{MP0} \vee \mathtt{MP1})))$$

Cette manière de représenter les sous-parties du système par les variables qu'elles contiennent peut être utilisée pour vérifier que (comme mentionné dans le cahier des charges) le remplissage des cuves peut être effectué en parallèle avec la distribution des briques. Il s'agit, là encore, d'une propriété d'atteignabilité, qui peut être vérifiée au moyen de la formule CTL suivante :

$$\mathsf{AG}\ (fdc \Rightarrow \mathsf{EF}\ (((\mathtt{VA} \vee \mathtt{VB} \vee \mathtt{VC}) \wedge \mathtt{MT}) \wedge fdc)$$

Franchissement des transitions On vérifie, en utilisant les variables représentant les étapes du SFC d'origine, que celles-ci sont franchies conformément aux règles d'évolution du SFC et aux valeurs des relations de transition. Par exemple, si l'étape `BRICK1` est active, elle ne cesse d'être active que si la valeur de la variable d passe à TRUE. On utilise pour cela la formule LTL suivante :

$$\mathsf{G}\,(fdc \Rightarrow ((\mathtt{BRICK1} \wedge \neg \mathtt{d}) \Rightarrow \mathsf{X}\,(\neg fdc\,\mathsf{U}\,(\mathtt{BRICK1} \wedge fdc))))$$

Cela signifie que, si l'étape BRICK1 est active mais que la valeur de d reste FALSE, l'étape BRICK1 restera active à la prochaine fin de cycle. On vérifie par là qu'il n'existe aucun autre moyen de désactiver l'étape que le respect de la condition de transition.

Correspondance étapes-actions Nous pouvons vérifier les correspondances entre l'activation d'une étape et l'action qu'elle entraîne, en vérifiant leur équivalence logique en fin de cycle :

$$\mathsf{G}\,(fdc \Rightarrow (\mathtt{START} \Leftrightarrow \mathtt{DONE}))$$
$$\mathsf{G}\,(fdc \Rightarrow (\mathtt{WEIGH\_A} \Leftrightarrow \mathtt{VA}))$$

Si une action est distribuée entre plusieurs étapes (activée par une étape, désactivée par une autre), nous pouvons vérifier qu'elle n'est pas déclenchée autrement que par ces étapes :

$$\mathsf{G}\,(fdc \Rightarrow ((\mathtt{BRICK1} \vee \mathtt{DROP\_1} \vee \mathtt{BRICK2}) \Leftrightarrow \mathtt{MT}))$$

Franchissement d'étapes avec hypothèses dynamiques Comment nous l'avons vu à la section 5.2.4.2 page 98, nous pouvons vérifier des formules de vivacité dépendant d'hypothèses dynamiques. Par exemple, nous pouvons vérifier que le remplissage du liquide B succède à celui du liquide A, sous réserve que le capteur de la balance enregistre la montée du liquide A jusqu'au niveau requis (nous n'avons pas codé dans notre modèle le fait que, quand une cuve est remplie, elle finit par être pleine... de même que nous supposons que les capteurs ont un fonctionnement normal). Nous pouvons par exemple vérifier cette propriété pr la formule

$$\begin{aligned} & \mathsf{G}\,(\mathtt{VA} \Rightarrow \mathsf{F}\,(\mathtt{POIDS\_A})) \\ \Rightarrow\quad & \mathsf{G}\,(\mathtt{WEIGH\_A} \Rightarrow \mathsf{F}\,(\mathtt{WEIGH\_B})) \end{aligned}$$

Nous pouvons vérifier des propriétés analogues pour toutes les franchissabilités qui requièrent des hypothèses dynamiques (la cuve finit par être pleine, les briques finissent par tomber quand le tapis roulant est déclenché, la grande cuve finit par arriver en bas si on la fait basculer, etc.).

7.2.3.3. Soumission à SMV

Le rapport de la vérification par SMV est donné à la section D.2 page 202. Il se compose de trois parties : une énumération de chaque formule et du temps nécessaire pour la vérifier (c'est le temps cumulé qui est indiqué), le résultat de la vérification pour chaque formule, et une récapitulation des ressources utilisées. Ici, toutes les formules sont vérifiées. Si l'une des formules n'est pas vérifiée, cela est indiqué dans le rapport, et un fichier contenant une trace est produit. SMV est livré avec un logiciel graphique, VW, qui est capable de lire cette trace et en fournir une représentation sous forme de tableau (en ligne les variables, en colonne les transitions). Cette représentation, et la fonctionnalité de pouvoir sélectionner les variables affichées, permet de retrouver l'erreur assez rapidement.

Le rapport donné pour le mixeur indique que la vérification est triviale : il a fallu moins de deux secondes pour vérifier la totalité des formules.

7.3. Chariot

7.3.1. Présentation

7.3.1.1. Description

Notre deuxième exemple concerne un chariot de manutention, se déplaçant le long d'un rail porteur. La partie que nous étudions se compose :

- d'un chariot animé par un moteur ;
- d'un pupitre, comportant :
  - un clavier permettant de saisir un numéro de position,
  - un voyant lumineux,
  - trois boutons-poussoirs.

La position du chariot sur le rail est indiquée 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`.

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.

7.3.1.2. Signaux

Signaux d'entrée

`position` (entier) position courante du chariot ;

`requête` (entier) valeur tapée au clavier ;

`go` (booléen) bouton de démarrage ;

`stop` (booléen) bouton d'arrêt ;

`ann_req` (booléen) bouton d'annulation de requête.

Signaux de sortie

`MG` (booléen) mouvement du moto-réducteur vers la gauche ;

`MD` (booléen) mouvement du moto-réducteur vers la droite ;

`VMC` (booléen) allumage du voyant chariot.

Variables internes

`mouvement` (booléen) indique que le chariot est en mouvement ;

`destination` (entier) destination, si une mission est en cours ;

`mission_en_cours` (booléen) indique si une mission est en cours.

7.3.2. Version IL

7.3.2.1. Programme

Une version du programme en IL est donnée ci-dessous.

```
TYPE
  POS : INT (0..31);
END_TYPE

VAR_INPUT
  position : POS;
  requete : POS;
  go : BOOL;
  stop : BOOL;
  ann_req : BOOL;
END_VAR

VAR_OUTPUT
  MG : BOOL := FALSE;
  MD : BOOL := FALSE;
  VMC : BOOL := FALSE;
```

```
END_VAR

VAR
  mouvement : BOOL := FALSE;
  destination : POS := 0;
  mission_en_cours : BOOL := FALSE;
END_VAR

              LD    stop
              JMPC  arret

              LD    destination
              EQ    position
              AND   mouvement
              JMPC  fin_mission

              LDN   mouvement
              ANDN  go
              AND   ann_req
              JMPC  fin_mission

              LDN   mouvement
              AND   go
              ANDN  ann_req
              RETCN

              LDN   mission_en_cours
              JMPCN marche

              LD    TRUE
              ST    mission_en_cours
              ST    VMC
              LD    requete
              ST    destination

marche:       LD    position
              EQ    destination
              RETC

              LD    TRUE
              ST    mouvement
              LD    position
              LT    destination
```

taille des entiers	2 bits	3 bits	4 bits	5 bits
temps de la vérification	19 s	41 s	12 min 39 s	53 min 35 s
noe uds BDD alloués	66 628	143 836	1 689 860	5 892 208

TAB. 7.1.: Temps de la vérification en fonction de la taille des variables entières.

```
              JMPC  droite

gauche:       LD    TRUE
              ST    MG
              RET

droite:       LD    TRUE
              ST    MD
              RET

fin_mission:  LD    FALSE
              ST    mission_en_cours
              ST    VMC

arret:        LD    FALSE
              ST    MG
              ST    MD
              ST    mouvement
```

7.3.2.2. Code SMV correspondant

Le code SMV correspondant au programme IL est donné à l'annexe E page 219.

7.3.2.3. Soumission à SMV

Le tableau 7.1 illustre la complexité grandissante de la vérification lorsque la taille des entiers augmente. Notre codage, qui ne procède à aucune optimisation particulière, permet néanmoins de vérifier l'exemple dans ses conditions de fonctionnement normales. Il est possible d'améliorer les performances de la vérification par des optimisations.

7.3.3. Version ST

7.3.3.1. Programme

Ci-après le programme ST implémentant le chariot.

```
IF NOT stop
THEN
```

```
    IF ((mouvement AND (destination = position))
        OR (NOT mouvement AND NOT go AND ann_req))
    THEN
      mission_en_cours := FALSE ;
      VMC := FALSE ;
    ELSE
      IF (NOT mouvement AND go AND (NOT ann_req))
      THEN
        IF NOT mission_en_cours THEN
          mission_en_cours := TRUE ;
          VMC := TRUE ;
          destination := requete ;
        END_IF ;
        IF destination = position
        THEN RETURN ;
        ELSE
          mouvement := TRUE ;
          IF position < destination
          THEN MD := TRUE ;
          ELSE MG := TRUE ;
          END_IF ;
          RETURN ;
        END_IF ;
      ELSE
        RETURN ;
      END_IF ;
    END_IF ;
  END_IF ;
  mouvement := 0 ;
  MG := 0 ;
  MD := 0 ;
```

7.3.3.2. Code SMV correspondant

L'exemple du chariot comporte des entiers bornés. Il est possible de considérer chacun des entiers concernés en les codant sous la forme de plusieurs booléens, puis traduire les opérateurs et appliquer la sémantique booléenne. L'exemple du chariot nous a permis de réaliser à titre expérimental une extension du modèle modulaire avec des modules s'appliquant aux entiers. Nous ne détaillerons pas cette approche, mais elle est clairement visible à la lecture du code SMV correspondant à l'exemple. Ce code SMV est présenté à l'annexe F page 225.

7.3.4. Propriétés

Après avoir vérifié, pour l'exemple précédent, des propriétés essentiellement structurelles, nous présentons ici des exemples de propriétés extraites du cahier des charges.

7.3.4.1. 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 vraies toutes les deux. Il s'agit d'un problème d'exclusion mutuelle, que nous exprimons l'aide de la formule suivante :

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

7.3.4.2. Respect de la commande d'arrêt

Nous vérifions que le moteur s'arrête effectivement en cas d'appui sur le bouton stop. Il faut bien entendu tenir compte du délai nécessaire à la réaction du système (le temps d'exécuter le programme). Les moteurs doivent normalement s'arrêter dès la fin de cycle suivant l'activation du bouton stop :

$$\mathsf{G}\ (\mathtt{stop} \Rightarrow (fdc \Rightarrow (\neg\mathtt{MG} \wedge \neg\mathtt{MD})))$$

Il serait possible de vérifier la formule suivante :

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

Cette formule 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.

7.3.4.3. 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 des deux formules (symétriques) suivantes :

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

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

Cette formule se lit (par exemple la première) « 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 (voir section 5.2.2 page 93), que nous exprimons à l'aide de l'opérateur U pour donner la formule suivante :

$$\mathsf{G}\ (\mathtt{MG} \Rightarrow ((\mathsf{G}\ \neg\mathtt{MD}) \vee (\neg\mathtt{MD}\ \mathsf{U}\ (\neg\mathtt{MG} \wedge \neg\mathtt{MD}))))$$

7.3.4.4. 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 :

$$\mathsf{G}\ ((\texttt{go} \wedge \neg\texttt{ann\_req}) \Rightarrow ((\texttt{MG} \vee \texttt{MD})\ \mathsf{W}\ (\texttt{stop} \vee (\texttt{position} = \texttt{destination}))))$$

Nous effectuons la traduction du W en U, mais, comme d'habitude, 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.

7.3.4.5. 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. Comme dans l'exemple précédent, nous pouvons formuler des hypothèses sur le comportement des entrées, et nous restreindre aux exécutions vérifiant ces hypothèses. Par exemple, nous pouvons formuler l'hypothèse suivante :

$$\mathsf{G}\ ((\texttt{MG} \vee \texttt{MD}) \Rightarrow \mathsf{F}\ (\texttt{stop} \vee (\texttt{position} = \texttt{destination})))$$

Cette hypothèse 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 à vérifier, en combinaison avec l'hypothèse qui précède, est alors :

$$\mathsf{G}\ ((\texttt{go} \wedge \neg\texttt{ann\_req}) \Rightarrow ((\texttt{MG} \vee \texttt{MD})\ \mathsf{U}\ (\texttt{stop} \vee (\texttt{position} = \texttt{destination}))))$$

7.4. Conclusion

Nous avons présenté deux exemples, issus de cas réels, ayant selon notre méthode fait l'objet d'une vérification. Malgré la taille non triviale de ces exemples, la vérification par SMV reste très performante (de l'ordre de quelques minutes pour l'ensemble des propriétés présentées). Il va de soi, cependant, que le codage modulaire engendre une complexité bien plus importante que les codages issus de la sémantique directe. Cela est principalement dû à la prolifération de variables qu'il provoque, et à la taille de la spécification obtenue. un travail d'optimisation du codage, où chaque simplification serait prouvée en regard de la sémantique opérationnelle, permettrait d'augmenter considérablement les performances de sa vérification, au point de les rapprocher des performances obtenues avec la sémantique directe.

8. Conclusion

Nous avons proposé des méthodes permettant d'effectuer la validation automatique de programmes écrits dans les langages de la norme IEC 61131-3. En particulier, les deux langages textuels IL et ST, définis par la norme, ont été étudiés. Les éléments constitutifs de l'approche décrite dans cette thèse sont les suivants :

1. définition d'une sémantique opérationnelle des langages ;
2. définition d'un codage de cette sémantique dans le format d'entrée d'un logiciel de model-checking symbolique (ici Cadence SMV) ;
3. écriture de propriétés dans le formalisme utilisé par le model-checker (ici les logiques temporelles LTL et CTL).

La définition de la sémantique permet de déduire, pour chaque programme IL ou ST, son modèle associé. Cette méthode est indépendante du produit utilisé en aval ; nous avons, pour notre étude de cas, utilisé SMV, qui exploite la technique prometteuse du model-checking symbolique. La définition du codage permet, par la suite, la mise en œuvre d'un compilateur (non présenté dans la thèse), qui pourra convertir automatiquement un programme donné en sa spécification associée dans un logiciel de validation (ici SMV).

Chacune des étapes menant du programme au model-checker est démontrable. La validité de la méthode peut, en outre, être directement perçue, du fait de la clarté et de la simplicité d'écriture de chacune des étapes. Dans le cas où ces opérations deviendraient plus complexes, par exemple dans le cas de la recherche d'une optimisation du code généré pour le model-checker, il serait nécessaire de mettre en place un formalisme permettant de valider, par la preuve, les optimisations du codage.

Notre approche est illustrée par le traitement d'exemples. On constate à l'expérimentation que, si la vérification s'avère faisable, et même triviale dans le cas de programmes booléens, la prise en compte des entiers engendre une forte complexité. La méthode de codage de la sémantique modulaire, présentée dans cette thèse, a besoin d'être optimisée. Cela peut être fait, en reliant par la preuve le code optimisé avec la sémantique opérationnelle.

D'autres extensions de la méthode peuvent être envisagées, comme par exemple la prise en compte de structures de données plus complexes, ou de l'exécution de processus en mode multi-tâches. Un compilateur de IL/ST vers SMV (ou un autre logiciel de model-checking), utilisant les techniques présentées dans cette thèse, peut être greffé à des environnements de programmation dédiés aux langages de la norme. Nous pensons que la méthode, par son fort degré d'automatisation et la relative simplicité de sa mise en œuvre, est pleinement applicable dans un contexte industriel, et peut être simplement ajoutée à des environnements existants, en offrant un meilleur niveau de validation.

Bibliographie

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Collection Études et Recherches en Informatique. Masson, Paris, 1992.

[BCK98a] I.D. Bates, E.G. Chester, et D.J. Kinniment. « Automatic generation of IEC 1131 PLC code from Statemate Statecharts ». Dans *Proc. 4th IEEE Workshop on Discrete Event Systems (WODES'98), Cagliari, Italy*, pages 167–172, août 1998.

[BCK98b] I.D. Bates, E.G. Chester, et D.J. Kinniment. « A case study in the automatic programming of a PLC based control system using Statemate Statecharts ». Dans *Proc. Control 98, University of Wales Swansea, UK*, volume 1, pages 832–837, septembre 1998.

[BHLL00a] S. Bornot, R. Huuck, Y. Lakhnech, et B. Lukoschus. « Utilizing Static Analysis for Programmable Logic Controllers ». Dans *ADPM 2000 : 4th International Conference on Automation of Mixed Processes : Hybrid Dynamic Systems, September 18-19, 2000, Dortmund, Germany*, 2000.

[BHLL00b] S. Bornot, R. Huuck, Y. Lakhnech, et B. Lukoschus. « Verification of Sequential Function Charts using SMV ». Dans *PDPTA 2000 : The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, June 26-29, 2000, Las Vegas, Nevada, USA*. CSREA Press, 2000.

[Bry86] R. E. Bryant. « Graph-based algorithms for boolean function manipulation ». *IEEE Transactions on Computers*, C-35(8) :677–691, 1986.

[Bry92] R. E. Bryant. « Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams ». *ACM Computing Surveys*, 24(3) :293–318, 1992.

[CBM90] O. Coudert, C. Berthet, et J. C. Madre. « Verification of Synchronous Sequential Machines Based on Symbolic Execution ». Dans *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89), Grenoble, June 1989*, volume 407 de *Lecture Notes in Computer Science*, pages 365–373. Springer, 1990.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, et Ph. Schnoebelen. « Towards the automatic verification of PLC programs written in Instruction List ». Dans *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA*, pages 2449–2454, octobre 2000.

[CCR01] G. Canet, S. Couffin, et O. Rossi. Validation dans le cadre de l'utilisation des SFC, LD et ST. Rapport final interne du projet Vulcain, mai 2001.

[CDP+00] G. Canet, B. Denis, A. Petit, O. Rossi, et Ph. Schnoebelen. « Un cadre pour la vérification automatique de programmes IL ». Dans *Actes 1re conf. int. francophone d'automatique (CIFA'2000), Lille, France*, pages 693–698. UCIS, Villeneuve d'Ascq, France, juillet 2000.

[CEI93] CEI (Commission électrotechnique internationale). « *NF EN 61131-3 : Automates programmables - partie 3* », 1993.

[CES86] E. M. Clarke, E. A. Emerson, et A. P. Sistla. « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications ». *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGH97] E. M. Clarke, O. Grumberg, et K. Hamaguchi. « Another Look at LTL Model Checking ». *Formal Methods in System Design*, 10(1) :47–71, 1997.

[CGL92] E. M. Clarke, O. Grumberg, et D. E. Long. « Model Checking and Abstraction ». Dans *Proc. 19th ACM Symp. Principles of Programming Languages, Albuquerque, NM, USA*, pages 343–354, janvier 1992.

[CGP99] E. M. Clarke, O. Grumberg, et D. A. Peled. *Model Checking*. The MIT Press, 1999.

[CHR91] Z. Chaochen, C. Hoare, et A. Ravn. « A calculus of duration ». *Information Processing Letters*, 40 :269–276, December 1991.

[CJ 98] CJ International. « *ISaGRAF, IEC 1131-3 Soft Logic, Guide Utilisateur* », 1998.

[dCR+00] O. de Smet, S. Couffin, O. Rossi, G. Canet, J.-J. Lesage, Ph. Schnoebelen, et H. Papini. « Safe programming of PLC using formal verification methods ». Dans *Proc. 4th Int. PLCopen Conf. on Industrial Control Programming (ICP'2000), Utrecht, The Netherlands*, pages 73–78. PLCOpen, Zaltbommel, The Netherlands, octobre 2000.

[DFMV98] H. Dierks, A. Fehnker, A. Mader, et F.W. Vaandrager. « Operational and logical semantics for polling real-time systems ». Dans A.P. Ravn et H. Rischel, éditeurs, *Proceedings of the Fifth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98), Lyngby, Denmark*, volume 1486 de *LNCS*, pages 29–40. Springer, 1998. Extended version available as Report CSI-R9813, Computing Science Institute, University of Nijmegen, April 1998.

[Die97a] H. Dierks. « PLC-Automata : A New Class of Implementable Real-Time Automata ». Dans M. Bertran et T. Rus, éditeurs, *ARTS'97*, volume 1231 de *LNCS*, pages 111–125. Springer, May 1997.

[Die97b] H. Dierks. « Synthesising Controllers from Real-Time Specifications ». Dans *Tenth International Symposium on System Synthesis*, pages 126–133. IEEE Computer Society, Sep 1997.

[DT00] H. Dierks et J. Tapken. « Modelling and Verifying of 'Cash-Point Service' Using Moby/PLC ». *Formal Aspects of Computing*, 12 :222–221, 2000.

[EFP94] Gottfried Egger, Andreas Fett, et Peter Pepper. « Formal Specification of a Safe PLC Language and its Compiler ». Dans *SafeComp'94 13th International Conference on Computer Safety, Reliability and Security*. ISA, 1994.

[EH86] E. A. Emerson et J. Y. Halpern. « "Sometimes" and "Not Never" Revisited : On Branching versus Linear Time Temporal Logic ». *Journal of the ACM*, 33(1) :151–178, 1986.

[Eme90] E. A. Emerson. Temporal and Modal Logic. Dans J. van Leeuwen, éditeur, *Handbook of Theoretical Computer Science, vol. B*, Chapitre 16, pages 995–1072. Elsevier Science Publishers, 1990.

[FL98] G. Frey et L. Litz. « Verification and Validation of Control Algorithms by Coupling of Interpreted Petri Nets ». Dans *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'98), San Diego, CA, USA, Oct. 1998*, volume 1, pages 7–12, 1998.

[FL00] G. Frey et L. Litz. « Formal methods in PLC programming ». Dans *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA, Oct. 2000*, pages 2431–2436, 2000.

[Fre00a] G. Frey. « Automatic Implementation of Petri net based Control Algorithms on PLC ». Dans *Proc. American Control Conference (ACC2000), Chicago, June 2000*, pages 2819–2823, 2000.

[Fre00b] G. Frey. « PLC Programming for Hybrid Systems via Signal Interpreted Petri Nets ». Dans *Proc. 4th Int. Conf. on Automation of Mixed Processes (ADPM), Dortmund, Germany, Sept. 2000*, pages 189–194, 2000.

[HM98a] M. Heiner et T. Menzel. « Instruction List Verification Using a Petri Net Semantics ». Dans *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'98), San Diego, CA, USA*, pages 716–721, octobre 1998.

[HM98b] M. Heiner et T. Menzel. « A Petri Net Semantics for the PLC Language Instruction List ». Dans *Proc. 4th IEEE Workshop on Discrete Event Systems (WODES'98), Cagliari, Italy*, pages 161–165, août 1998.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Int., 1991.

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

[JEK+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, et L.J. Hwang. « Symbolic Model Checking : 10^{20} States and Beyond ». *Information and Computing*, 98(2) :142–170, 1992.

[JF01] F. Jiménez-Fraustro. « *Conception sûre des automates industriels : modélisation synchrone de langages d'automates programmables de la norme CEI-61131-3* ». Thèse de doctorat, Université de Rennes I, mars 2001.

[JFR99a] F. Jimenéz-Fraustro et É. Rutten. « Modélisation synchrone de standards de programmation de systèmes de contrôle : le langage ST de la norme CEI 1131-3 ». Dans *Journées d'études « Nouvelles percées dans les langages pour l'automatique », SEE – Club 18 (Automatique et automatisation industrielle), Amiens (France)*, novembre 1999.

[JFR99b] F. Jiménez-Fraustro et E. Rutten. « A synchronous model of the PLC programming language ST ». Dans *Proc. 11th Euromicro Conference on Real-Time Systems (ECRTS'99), York, UK*, pages 21–24, June 1999.

[JFR00] F. Jimenez-Fraustro et E. Rutten. « Hybrid simulation of IEC-61131 PLC programs using Signal and Simulink ». Dans *Proc. 4th Int. Conf. on Automation of Mixed Processes (ADPM), Dortmund, Germany, Sept. 2000*, pages 171–176, 2000.

[KES99] S. Kowalewski, S. Engell, et O. Stursberg. « Verification of Logic Controllers for Continuous Plants ». Dans *Advances in Control : Highlights of ECC'99*, pages 345–389. Springer-Verlag, 1999.

[Lar95] F. Laroussinie. « About the expressive power of CTL combinators ». *Information Processing Letters*, 54(6) :343–345, 1995.

[LCRRL99] S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, et J.-J. Lesage. « Formal validation of PLC programs : a survey ». Dans *Proc. European Control Conf. (ECC'99), Karlsruhe, Germany*, septembre 1999. Actes sur CD-ROM, communication 741.

[Lew98] R. W. Lewis. *Programming industrial control systems using the IEC 1131-3 (Revised edition)*. The Institution of Electrical Engineers, 1998. ISBN 0-85296-950-3.

[LPY97] K. G. Larsen, P. Pettersson, et Wang Yi. « UPPAAL in a Nutshell ». *Journal of Software Tools for Technology Transfer*, 1(1–2) :134–152, 1997.

[Mad00] Angelika Mader. « A Classification of PLC Models and Applications ». Dans *Proceedings 5th Workshop on Discrete Event Systems (WODES2000),* August 21-23, 2000, Gent, Belgium. Kluwer Academic Publishers, 2000.

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

[McM93] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[Mel01] Phil Melore. « Your Personal PLC tutor ». Page Web, 1996-2001. `http://www.plcs.net/`.

[Met] Meta Software Corporation. « *Design/CPN Reference Manual for X-Windows* ». `http://www.daimi.au.dk/designCPN/man/Reference/Reference.All.pdf`.

[MM00] Th. MERTKE et Th. MENZEL. « Methods and tools to the verification of safety-related control software ». Dans *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA, Oct. 2000*, 2000.

[Moo94] I. MOON. « Modeling Programmable Logic Controllers for Logic Verification ». *IEEE Control Systems*, 14(2) :53–59, 1994.

[MW99] Angelika MADER et Hanno WUPPER. « Timed Automaton Models for Simple Programmable Logic Controllers ». Dans *Proceedings of Euromicro Conference on Real-Time Systems 1999*, York, UK, june, 9-11 1999. to appear.

[MW00] Angelika MADER et Hanno WUPPER. « What is the formal method for PLC applications ? ». Dans *Proceedings ADPM 2000 : 4th International Conference on Automation of Mixed Processes : Hybrid Dynamic Systems,* September 18-19, 2000, Dortmund, Germany, 2000.

[Old99] Ernst-Rüdiger OLDEROG. Correct Real-time Software for Programmable Logic Controllers. Dans *Lecture Notes in Computer Science*, numéro 1710, pages 342–362. Springer-Verlag, 1999.

[Pix92] C. PIXLEY. « A Theory and Implementation of Sequential Hardware Equivalence ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(12) :1469–1478, 1992.

[RK98] M. RAUSCH et B. KROGH. « Formal verification of PLC programs ». Dans *Proc. American Control Conference (ACC98), Philadelphia, PA, USA*, pages 234–238, 1998.

[RS00] O. ROSSI et Ph. SCHNOEBELEN. « Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs ». Dans *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 OTHERS. *Vérification de logiciels : Techniques et outils du model-checking.* Vuibert, 1999.

[SMV] « Archives de la liste de distribution smv-users ». Page web. `http://www-cad.eecs.berkeley.edu/~kenmcmil/hypermail/smv-users/`.

[Tou97] K TOURLAS. « An Assessment of the IEC 1131 -3 Standard on Languages for Programmable Controllers ». Dans Peter DANIEL, éditeur, *SAFECOMP97 : the 16th International Conference on Computer Safety, Reliability and Security York, UK, 7-10 September 1997*, pages 210–219. Springer, 1997.

[Var96] M. Y. VARDI. « An Automata-Theoretic Approach to Linear Temporal Logic ». *Lecture Notes in Computer Science*, 1043 :238– ? ?, 1996.

[VW86] M. VARDI et P. WOLPER. « An Automata-Theoretic Approach to Automatic Program Verification ». pages 332–344, 1986.

[Yov97] S. Yovine. « Kronos : A Verification Tool for real-Time Systems ». *Journal of Software Tools for Technology Transfer*, 1(1–2) :123–133, 1997.

Table des figures

Liste des tableaux

A. Grammaire concrète des langages

A.1. Introduction

La norme [CEI93, IEC93] fournit, dans son annexe B, une grammaire complète des langages textuels. Nous définissons pour notre part un sous-ensemble de cette grammaire, qui correspond aux fragments de langages significatifs que nous étudions. Notre grammaire suit les mêmes conventions que celle de la norme. Après avoir décrit ces conventions, nous émettrons quelques remarques sur la grammaire fournie par la norme ; nous donnons ensuite la grammaire complète des langages que nous considérons.

A.2. Format de la grammaire

A.2.1. Conventions normatives d'écriture des grammaires concrètes

L'annexe A de la norme définit les conventions utilisées pour la description des grammaires concrètes. Les grammaires que nous présentons dans ce chapitre sont conformes aux conventions de la norme. Ces conventions peuvent se résumer comme suit :

- les symboles terminaux textuels sont représentés entre apostrophes (`'`) : `'ABC'` représente la chaîne de caractères `ABC` apparaissant dans un programme[1]. Le délimiteur de fin de ligne, qui a une valeur syntaxique en IL, est désigné par `EOL`. La « chaîne vide » est représentée par `NIL` ;
- une règle de grammaire s'écrit sous la forme `symbole_non_terminal ::= suite de symboles`, où la suite de symboles est telle que :

  1. la suite de symboles contient des terminaux, des non-terminaux, ou `NIL`, ou `EOL`,
  2. `S1 S2` signifie « `S1` suivi de `S2` »,
  3. `S1 | S2` signifie un choix entre `S1` et `S2`,
  4. `(S)` est équivalente à `S` (simple parenthésage),
  5. `{S}` dénote une séquence répétée de `S`, éventuellement vide,
  6. `[S]` dénote une occurrence optionnelle de S (équivalent à `(S | NIL)`).

[1] En fait, la norme propose aussi les guillemets typographiques (`"`) pour pouvoir désigner la chaîne `'` par `"'"` dans trois lignes de la grammaire. Nous n'utilisons pas cette possibilité ici.

```
single_element_type_name ::= simple_type_name
                           | subrange_type_name
                           | enumerated_type_name
simple_type_name ::= identifier
subrange_type_name ::= identifier
enumerated_type_name ::= identifier
```

FIG. A.1.: Exemple de conflit dans la grammaire fournie par la norme.

Le parenthésage est prioritaire ; de plus, la séquence a priorité sur l'alternative, c'est-à-dire que `S1 | S2 S3` est équivalent à `S1 | (S2 S3)`. On ne tient pas compte des caractères d'espacement (espace, tabulation). Le retour à la ligne a une importance syntaxique uniquement pour les corps de programmes IL. Dans les autres cas (déclarations de types et de variables, corps de programme ST), le retour à la ligne n'est pas pris en compte et peut être placé n'importe où.

A.2.2. Remarques sur la grammaire fournie par la norme

La grammaire présentée à l'annexe B de la norme est ambiguë. La figure A.1 illustre ce problème par un exemple typique, directement extrait de la norme (section B.1.3.3).

Dans un cas tel que celui de la figure A.1, un compilateur qui lit dans son entrée un `identifier` ne peut pas décider laquelle des trois réductions possibles il doit adopter. Ici, il suffirait de corriger la grammaire en écrivant la règle :

```
single_element_type_name ::= identifier
```

On retrouve la même erreur, de manière moins directe que dans cet exemple, à divers endroits de la grammaire donnée par la norme : une distinction est souvent faite entre des identificateurs, ayant la même définition syntaxique et pouvant être placés aux mêmes endroits, mais représentant des objets différents[2].

Notre grammaire concrète présente une version non ambiguë des langages.

A.3. Éléments communs

A.3.1. Lettres, chiffres et identificateurs

```
letter ::= 'A' | 'B' | <...> | 'Z'
         | 'a' | 'b' | <...> | 'z'
digit ::= '0' | '1' | '2' | '3' | '4'
        | '5' | '6' | '7' | '8' | '9'
```

[2]L'erreur ici commise est de confondre les noms des non-terminaux, choisis pour faciliter la lecture de la grammaire, avec la sémantique des éléments de langage qu'ils représentent. Un compilateur ne peut pas décider ce que *signifie* un identificateur.

```
identifier ::= (letter | ('_' (letter | digit)))
                   {['_'] (letter | digit)}
```

A.3.2. Constantes

```
constant ::= numeric_literal | boolean_literal
numeric_literal ::= integer_literal
integer_literal ::= signed_integer
signed_integer ::= [ '+' | '-' ] integer
integer ::= digit { digit }
boolean_literal ::= 'TRUE' | 'FALSE'
```

A.3.3. Types de données

```
data_type_name ::= non_generic_type_name
non_generic_type_name ::= elementary_type_name
                          | derived_type_name
```

A.3.3.1. Types de données élémentaires

```
elementary_type_name ::= numeric_type_name
                         | bit_string_type_name
numeric_type_name ::= integer_type_name
integer_type_name ::= signed_integer_type_name
                      | unsigned_integer_type_name
signed_integer_type_name ::= 'SINT' | 'INT'
                             | 'DINT' | 'LINT'
unsigned_integer_type_name ::= 'USINT' | 'UINT'
                               | 'UDINT' | 'ULINT'
bit_string_type_name ::= 'BOOL'
```

A.3.3.2. Types de données dérivés

```
derived_type_name ::= identifier
data_type_declaration ::= 'TYPE' type_declaration ';'
                          {type_declaration ';'}
                          'END_TYPE'
type_declaration ::= simple_type_declaration
                     | subrange_type_declaration
simple_type_declaration ::= identifier
                            ':' simple_spec_init
simple_spec_init ::= simple_specification
                     [ ':=' constant ]
```

```
simple_specification ::= elementary_type_name
                        | identifier
subrange_type_declaration ::= identifier
                              ':' subrange_spec_init
subrange_spec_init ::= subrange_specification
                      [ ':=' signed_integer ]
subrange_specification ::= integer_type_name
                          '(' subrange ')'
subrange ::= signed_integer '..' signed_integer
```

A.3.3.3. Déclaration de variables

```
variable ::= identifier
input_declarations ::= 'VAR_INPUT'
                      var_init_decl ';'
                      {var_init_decl ';'}
                      'END_VAR'
var_init_decl ::= var1_list ':'
                 (simple_spec_init | subrange_spec_init)
var1_list ::= identifier
             {',' identifier}
output_declarations ::= 'VAR_OUTPUT'
                       var_init_decl ';'
                       {var_init_decl ';'}
                       'END_VAR'
input_output_declarations ::= 'VAR_INT_OUT'
                             var_init_decl ';'
                             {var_init_decl ';'}
                             'END_VAR'
var_declarations ::= 'VAR'
                    var_init_decl ';'
                    {var_init_decl ';'}
                    'END_VAR'
```

A.4. Langage IL

A.4.1. Instructions et opérandes

```
instruction_list ::= il_instruction
                    {il_instruction}
il_instruction ::= [label] ':'
                  [   il_simple_operation
                    | il_expression
```

```
                    | il_jump_operation
                    | il_return_operator ]
                  EOL {EOL}
label ::= identifier
il_simple_operation ::= il_simple_operator [il_operand]
il_expression ::= il_expr_operator '('
                 [il_operand] EOL {EOL}
                 [simple_instr_list]
                 ')'
il_jump_operation ::= il_jump_operator label
il_operand ::= constant | variable
simple_instr_list ::= il_simple_instruction
                     {il_simple_instruction}
il_simple_instruction ::= (  il_simple_operation
                           | il_expression)
                         EOL {EOL}
```

A.4.2. Opérateurs

```
il_simple_operator ::= 'LD' | 'LDN' | 'ST' | 'STN'
                     | 'NOT' | 'S' | 'R'
                     | il_expr_operator
il_expr_operator ::= 'AND' | 'OR' | 'XOR'
                   | 'ANDN' | 'ORN' | 'XORN'
                   | 'ADD' | 'SUB' | 'MUL' | 'DIV' | 'MOD'
                   | 'GT' | 'GE' | 'EQ'
                   | 'LT' | 'LE' | 'NE'
il_return_operator ::= 'RET' | 'RETC' | 'RETCN'
il_jump_operator ::= 'JMP' | 'JMPC' | 'JMPCN'
```

A.5. Langage ST

A.5.1. Expressions

```
expression ::= xor_expression {'OR' xor_expression}
xor_expression ::= and_expression {'XOR' and_expression}
and_expression ::= comparison {('&' | 'AND') comparison}
comparison ::= equ_expression { ('=' | '<>') equ_expression}
equ_expression ::= add_expression
                  {comparison_operator add_expression}
comparison_operator ::= '<' | '>' | '<=' | '>='
add_expression ::= term {add_operator term}
```

```
add_operator ::= '+' | '-'
term ::= power_expression
         {multiply_operator power_expression}
multiply_operator ::= '*' | '/' | 'MOD'
power_expression ::= unary_expression
                     {'**' unary_expression}
unary_expression ::= [unary_operator] primary_expression
unary_operator ::= '-' | 'NOT'
primary_expression ::= constant | variable
                       | '(' expression ')'
                       | function_name
                         '(' param_assignment
                         {',' param_assignment}
                         ')'
```

A.5.2. Énoncés

```
statement_list ::= statement ';' {statement ';'}
statement ::= NIL | assignment_statement
              | selection_statement | iteration_statement
assignment_statement ::= variable ':=' expression
selection_statement ::= if_statement | case_statement
if_statement ::= 'IF' expression 'THEN' statement_list
                   {'ELSIF' expression 'THEN' statement_list}
                   ['ELSE' statement_list]
                 'END_IF'
case_statement ::= 'CASE' expression 'OF' case_element
                     {case_element}
                     ['ELSE' statement_list]
                   'END_CASE'
case_element ::= case_list ':' statement_list
case_list ::= case_list_element {',' case_list_element}
case_list_element ::= subrange | signed_integer
iteration_statement ::= for_statement | while_statement
                        | repeat_statement | exit_statement
for_statement ::= 'FOR' control_variable ':=' for_list
                  'DO' statement_list 'END_FOR'
control_variable ::= identifier
for_list ::= expression 'TO' expression ['BY' expression]
while_statement ::= 'WHILE' expression
                    'DO' statement_list 'END_WHILE'
repeat_statement ::= 'REPEAT' statement_list
                     'UNTIL' expression 'END_REPEAT'
```

```
exit_statement ::= 'EXIT'
```

B. Sémantique IL directe étendue au type d'entiers bornés

B.1. Introduction

Nous présentons ici la sémantique directe du langage IL, comportant une prise en compte des entiers bornés. Cette sémantique a été présentée dans [CDP+00] et [CCL+00]. Pour la définition de la sémantique booléenne, on se reportera à la section 4.2.3 page 49.

B.2. Types et variables

La déclaration d'une variable x est la donnée d'un symbole s désignant la variable, de son type $\mathrm{t}(x)$ et d'une valeur initiale optionnelle $\mathrm{init}(x) = c^i$. 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 $\mathtt{minint} \leq min_t \leq max_t \leq \mathtt{maxint}$ où `minint` et `maxint` sont des constantes.

On rappelle qu'une valuation de Σ est une fonction associant à chaque symbole de variable s la valeur courante de la variable qu'il représente. Soit x la variable représentée par le symbole s. Ici, les valeurs sont `TRUE` ou `FALSE` 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 t.

B.3. Système de transitions

On représente le comportement d'un programme IL bien formé par un système de transitions $A = \langle Q, \rightarrow, I\rangle$, où l'ensemble des états (aussi appelés *configurations*) Q est constitué des triplets de la forme (V, a, m), tel que :

- V est une valuation de Σ ;
- $a \in \{\mathtt{FALSE}, \mathtt{TRUE}\} \cup [\mathtt{minint}, \mathtt{maxint}]$ est la valeur de l'accumulateur ;
- $m \in \{1, \ldots, k+1\} \cup \{err\}$ est le compteur de programme. La valeur err du compteur de programme représente une situation d'erreur ; elle est utilisée pour les calculs qui ne respectent pas les règles de typage.

$I \subseteq Q$ est l'ensemble des *états initiaux* : (V, a, m) est initial si et seulement si le compteur de programme m vaut $k + 1$ et si, pour tout variable (s, c^i), $V(s) = c^i$.

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

B.3.1. Définition des transitions

Il reste à définir quelles transitions existent entre les états de Q (le composant $\rightarrow \subseteq Q \times Q$ de A). À cet effet, 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 justement sur la ligne i. En général, l'état courant sera modifié en fonction de la nature de l'instruction et on passera à la ligne $i+1$. Nous notons ins_i l'instruction de la ligne i du programme IL.

Nous dirons que deux variables sont de types *compatibles* si elles sont toutes les deux de type entier borné (même si leurs bornes diffèrent) ou toutes les deux de type booléen. Nous dirons que deux variables sont *incompatibles* si une est de type entier et l'autre de type booléen. On considèrera l'accumulateur comme étant de type booléen si $a \in \{\texttt{FALSE}, \texttt{TRUE}\}$, de type entier sinon.

B.3.1.1. Opérateurs d'affectation

Dans le cas $ins_i = \texttt{ST}\,s$, pour tout a et pour toute valuation V :

- si s et a sont de types incompatibles, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si s et a sont booléens, on crée une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = a$;
- si s et a sont de type entier borné (et $\mathrm{t}(s)$ est le type de s) :
  - si $a \in [min_{\mathrm{t}(s)}, max_{\mathrm{t}(s)}]$, on crée une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = a$;
  - si $a \notin [min_{\mathrm{t}(s)}, max_{\mathrm{t}(s)}]$, on crée une transition $(V, a, i) \rightarrow (V, a, err)$.

Dans le cas $ins_i = \texttt{LD}\,s$, pour tout a et pour toute valuation V, on crée une transition $(V, a, i) \rightarrow (V, V(s), i+1)$. On note que cette opération modifie l'accumulateur et par conséquent modifie également son type.

Dans le cas $ins_i = \texttt{LDN}\,s$, pour tout a et pour toute valuation V :

- si s n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si s est booléen, on crée une transition $(V, a, i) \rightarrow (V, \neg V(s), i+1)$.

Dans le cas $ins_i = \texttt{STN}\,s$, pour tout a et pour toute valuation V :

- si a ou s n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a et s sont tous les deux de type booléen, on crée une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = \neg a$.

Dans le cas $ins_i = \texttt{S}\,s$, pour tout a et pour toute valuation V :

- si a ou s n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a et s sont tous les deux de type booléen, on crée une transition $(V, a, i) \rightarrow (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) = \texttt{TRUE}$ si $a = \texttt{TRUE}$ et $V'(s) = V(s)$ sinon.

Dans le cas $ins_i = \texttt{R}\,s$, pour tout a et pour toute valuation V :

Opérateur *op*	a'
`GT`	`TRUE` si $a > V(s)$, `FALSE` sinon
`GE`	`TRUE` si $a \geq V(s)$, `FALSE` sinon
`EQ`	`TRUE` si $a = V(s)$, `FALSE` sinon
`NE`	`TRUE` si $a \neq V(s)$, `FALSE` sinon
`LE`	`TRUE` si $a \leq V(s)$, `FALSE` sinon
`LT`	`TRUE` si $a < V(s)$, `FALSE` sinon

TAB. B.1.: Valeur de l'accumulateur après une instruction de comparaison, pour tout a et pour tout s.

Opérateur *op*	$\mathrm{b}_{op}(a, s)$
`AND`	$a \wedge V(s)$
`ANDN`	$a \wedge \neg V(s)$
`&`	$a \wedge V(s)$
`&N`	$a \wedge \neg V(s)$
`OR`	$a \vee V(s)$
`ORN`	$a \vee \neg V(s)$
`XOR`	$(\neg a \wedge V(s)) \vee (a \wedge \neg V(s))$
`XORN`	$(\neg a \wedge \neg V(s)) \vee (a \wedge V(s))$

TAB. B.2.: Valeur de l'accumulateur après une instruction sur les booléens, pour tout a et pour tout s.

- si a ou s n'est pas de type booléen, on crée une transition $(V, a, i) \to (V, a, err)$;
- si a et s sont tous les deux de type booléen, on crée une transition $(V, a, i) \to (V', a, i+1)$ avec $V'(x) = V(x)$ si $x \neq s$, $V'(s) =$ `FALSE` si $a =$ `TRUE` et $V'(s) = V(s)$ sinon.

B.3.1.2. Opérateurs de comparaison

Dans le cas où $ins_i = op\, s$ et *op* est un opérateur de comparaison, pour tout a et pour toute valuation V :

- si s et a sont de types incompatibles, on crée une transition $(V, a, i) \to (V, a, err)$;
- si s et a sont de types compatibles, on crée une transition $(V, a, i) \to (V, a', i+1)$, où a' dépend de *op*, a et s, et est défini comme décrit par le tableau B.1.

B.3.1.3. Opérateurs sur les booléens

Dans le cas où $ins_i = op\, s$ et *op* est un opérateur sur les booléens, pour tout a et pour toute valuation V :

- si a ou s n'est pas de type booléen, on crée une transition $(V, a, i) \to (V, a, err)$;
- si a et s sont tous les deux de type booléen, on crée une transition $(V, a, i) \to (V, \mathrm{b}_{op}(a, s), i+1)$, où b_{op} est défini comma décrit par le tableau B.2.

Opérateur op	$\mathrm{f}_{op}(a,s)$
`ADD`	$a + V(s)$
`SUB`	$a - V(s)$
`MUL`	$a \cdot V(s)$
`DIV`	$a/V(s)$

TAB. B.3.: Valeur de l'accumulateur après une instruction sur les entiers, pour tout a et pour tout s.

B.3.1.4. Opérateurs sur les entiers

Dans le cas où $ins_i = op\, s$ et op est un opérateur sur les entiers bornés, pour tout a et pour toute valuation V :

- si a ou s n'est pas de type entier borné, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a et s sont tous les deux de type entier borné, soit $\mathrm{f}_{op}(a, s)$ tel que défini par le tableau B.3. on crée une transition :
  - $(V, a, i) \rightarrow (V, a, err)$ si $\mathrm{f}_{op}(a, s) \notin [\mathtt{minint}, \mathtt{maxint}]$;
  - $(V, a, i) \rightarrow (V, \mathrm{b}_{op}(a, s), i + 1)$ si $\mathrm{f}_{op}(a, s) \in [\mathtt{minint}, \mathtt{maxint}]$.

B.3.1.5. Opérateurs de saut

Dans le cas où $ins_i = \mathtt{JMP}\, l$ avec l étiquette de la ligne i_l, on crée une transition $(V, a, i) \rightarrow (V, a, i_l)$.

Dans le cas où $ins_i = \mathtt{JMPC}\, l$ avec l étiquette de la ligne i_l, pour tout a et pour toute valuation V :

- si a n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a est de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, i_l)$ si $a = \mathtt{TRUE}$ et on crée une transition $(V, a, i) \rightarrow (V, a, a + 1)$ si $a = \mathtt{FALSE}$.

Dans le cas où $ins_i = \mathtt{JMPCN}\, l$ avec l étiquette de la ligne i_l, pour tout a et pour toute valuation V :

- si a n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a est de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, i_l)$ si $a = \mathtt{FALSE}$ et on crée une transition $(V, a, i) \rightarrow (V, a, a + 1)$ si $a = \mathtt{TRUE}$.

Dans le cas où $ins_i = \mathtt{RET}$, on crée une transition $(V, a, i) \rightarrow (V, a, k)$.

Dans le cas où $ins_i = \mathtt{RETC}$, pour tout a et pour toute valuation V :

- si a n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a est de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, k)$ si $a = \mathtt{TRUE}$ et on crée une transition $(V, a, i) \rightarrow (V, a, a + 1)$ si $a = \mathtt{FALSE}$.

Dans le cas où $ins_i = \mathtt{RETCN}$, pour tout a et pour toute valuation V :

- si a n'est pas de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, err)$;
- si a est de type booléen, on crée une transition $(V, a, i) \rightarrow (V, a, k)$ si $a = \mathtt{FALSE}$ et on crée une transition $(V, a, i) \rightarrow (V, a, a + 1)$ si $a = \mathtt{TRUE}$.

B.3.2. Erreur

L'état d'erreur est bloquant. Comme nous cherchons à obtenir un système de transitions sans état terminé, nous modélisons le blocage par une boucle infinie dans l'état d'erreur : pour tous V et a on crée une transition $(V, a, err) \to (V, a, err)$.

B.3.3. Lecture des entrées

Enfin on crée des transitions modélisant la phase de lecture des entrées, qui intervient juste après la fin d'un cycle de l'API. Nous considérons que les nouvelles valeurs des entrées peuvent prendre des valeurs arbitraires (respectant le typage) : pour toutes valuations V et V' ne différant que sur leurs variables d'entrées (c.-à-d. si $x \in \Sigma \setminus \Sigma_{in}$, alors $V'(x) = V(x)$) on crée une transition de $(V, a, k{+}1)$ vers $(V', a, 1)$. Ces transitions permettent également de simuler le nouveau cycle (voir la description du cycle de l'API à la section 1.2.1.1 page 13).

B.3.4. Déterminisme

On observera que pour la relation $\to$ 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 un non-déterminisme.

C. Programme Perl de conversion des programmes IL_0 en SMV

On trouvera ci-après un programme de conversion des programmes IL en SMV, d'après la sémantique directe. Ce programme n'est pas destiné à fournir une définition précise de la conversion (dont on trouvera une description à la section 6.2 page 119). Le lecteur pourra apprécier, par la donnée complète de ce court programme, la simplicité de la traduction.

```
#!/usr/local/bin/perl
#! Transformateur IL0 -> SMV

@types_variables = ("entree", "sortie", "entree_sortie", "local");
# Le type correspondant à « entrée » doit etre le premier du tableau.
%types_trans = (
                "bool" => "boolean"
               );

%etiquette;

# Compteur de programme
$pc = 1;

MAIN : {
  $fichier=<@ARGV>;

  chomp ($fichier);
  $fichier or die "Donnez le nom du fichier il0 en argument.\n";

  open (IL0, $fichier) or die "Problème d'ouverture du fichier : $!\n";
  open (SMV, ">" . $fichier . ".smv")
    or die "Problème d'ouverture du fichier : $!\n";

  while ($ligne = <IL0>) {

    chomp ($ligne);

    $ligne =~ s { \(\* .*? \*\) } []gsx; # Suppression des commentaires
    $ligne =~ s/^\s*(.*?)\s*$/$1/; # Élimination des espaces en début de ligne
    unless ($ligne) {
      next;
    }
```

```
    if ($ligne =~ /VAR/) {
      lit_decl ($ligne) ;
    }
    else {
      lit_source ($ligne) ;
    }
  }

  ecrit_decl () ;
  ecrit_init () ;
  ecrit_transitions () ;
  ecrit_formules () ;
}

sub lit_decl ($ligne)  {

  my ($variable, $type, $initial, $type_var) ;
  my (@decl_table) ;

  chomp ($ligne) ;
  if ($ligne eq "VAR_INPUT") {
    $type_var = $types_variables[0] ;
  } elsif ($ligne eq "VAR_OUTPUT") {
    $type_var = $types_variables[1] ;
  } elsif ($ligne eq "VAR_IN_OUT") {
    $type_var = $types_variables[2] ;
  } elsif ($ligne eq "VAR") {
    $type_var = $types_variables[3] ;
  } else {
    die "Déclaration de variable inconnue : $ligne\n" ;
  }

  while ($ligne = <IL0>) {
    if ($ligne =~ /END_VAR/) {
      last ;
    }
    die "Désolé, je n'accepte pas les déclarations multiples.\n"
      if ($ligne =~ /,/) ;

    $ligne =~ s { \(\* .*? \*\) } []gsx ; # Suppression des commentaires
    $ligne =~ s/^\s*(.*?)\s*$/$1/ ; # Élimination des espaces en début de ligne
    chomp ($ligne) ;

    $initial = 0 ;
    ($variable, $type, $initial) = split (/[ :;=]+/, $ligne) ;

    die "La variable $variable n'a pas de type\n"
      unless ($type) ;

    $decl_var{$type_var} .= $variable . " " ;
```

```
    $types_var{$variable} = $type ;
    $init_var{$variable} = $initial ;
  }
}

sub lit_source  ($ligne) {

  my ($label, $instruction, $argument) ;

  $pc++ ;

  if ($ligne =~ / :/) {
    ($label, $instruction, $argument) = split (/[ :\t]+/, $ligne) ;
    $etiquette{$label} = $pc ;
  } else {
    ($instruction, $argument) = split (/[ \t]+/, $ligne) ;
  }

  $instructions[$pc] = $instruction ;
  $arguments[$pc] = $argument ;
}

sub ecrit_decl {

  print SMV "#define FDC (pc = $pc)\n" ;
  print SMV "#define TRUE 1\n" ;
  print SMV "#define FALSE 0\n\n" ;

  print SMV "module main ()\n{\n" ;

  foreach $type_var (@types_variables) {
    if ($decl_var{$type_var}) {

      my (@io_type) = split (/ /, $decl_var{$type_var}) ;

      print SMV "/* Variables : $type_var */\n" ;

      foreach $variable (@io_type) {
        my ($type_il0) = $types_var{$variable} ;
#       my ($type_smv) = $types_trans{$type_il0} ;
        my ($type_smv) = "boolean" ;

        print SMV "  $variable\t : $type_smv ;\n" ;
      }
    }
  }
  print SMV "\n" ;

  print SMV "# Variables internes\n" ;
  print SMV "  pc\t : 1 .. $pc ;\n" ;
  print SMV "  acc\t : boolean ;\n" ;
```

```
  print SMV "  Lock\t : boolean ;\n\n" ;
}

sub ecrit_init {

  print SMV "  init (pc) := 1 ;\n" ;

  foreach $type_var (@types_variables) {
    if ($decl_var{$type_var}) {

      my (@io_type) = split (/ /, $decl_var{$type_var}) ;

      foreach $variable (@io_type) {
        my ($init) = $init_var{$variable} ;
        print SMV "  init ($variable) := $init ;\n" unless ($init eq "") ;
      }
    }
  }
  print SMV "\n\n" ;
}

sub ecrit_transitions {

  my (%store, %set, %reset) ;
  my (@jmp) ;

  # Acc

  print SMV "  next(acc) := case\n{\n" ;
  for ($i = 1 ; $i <= $pc ; $i++)
    {
      my ($instruction) = $instructions[$i] ;
      my ($argument) = $arguments[$i] ;

      if ($instruction eq "")
        {
          next ;
        }
      elsif ($instruction eq "LD")
        {
          print SMV "    pc = " . ($i - 1) . "\t : $argument ;\n" ;
        }
      elsif ($instruction eq "LDN")
        {
          print SMV "    pc = " . ($i - 1) . "\t : !$argument ;\n" ;
        }
      elsif ($instruction eq "AND")
        {
          print SMV "    pc = " . ($i - 1) . "\t : acc & $argument ;\n" ;
        }
      elsif ($instruction eq "ANDN")
```

```
      {
        print SMV "    pc = " . ($i - 1) . "\t : acc &!$argument ;\n" ;
      }
    elsif ($instruction eq "OR")
      {
        print SMV "    pc = " . ($i - 1) . "\t : acc | $argument ;\n" ;
      }
    elsif ($instruction eq "ORN")
      {
        print SMV "    pc = " . ($i - 1) . "\t : acc |!$argument ;\n" ;
      }
    elsif ($instruction eq "ST")
      {
        die "La variable $argument n’a pas été déclarée\n"
          unless ($types_var{$argument}) ;

        $store{$argument} .= $i . " " ;
      }
    elsif ($instruction eq "S")
      {
        die "La variable $argument n’a pas été déclarée\n"
          unless ($types_var{$argument}) ;

        $set{$argument} .= $i . " " ;
      }
    elsif ($instruction eq "R")
      {
        die "La variable $argument n’a pas été déclarée\n"
          unless ($types_var{$argument}) ;

        $reset{$argument} .= $i . " " ;
      }
    elsif ($instruction =~ "JMP")
      {
        @jmp = (@jmp, $i) ;
      }
    elsif ($instruction =~ "RET")
      {
        @jmp = (@jmp, $i) ;
      }
  }

print SMV "    default\t : acc ;\n} ;\n\n" ;

# pc

print SMV "  next(pc) := case\n{\n" ;

foreach $saut (@jmp) {
  my ($commande) = $instructions[$saut] ;
  my ($label) = $arguments[$saut] ;
```

```
      my ($destination) = $etiquette{$label};

      if ($commande eq "JMP")
        {
          print SMV  "    pc = " . ($saut-1) . "\t\t : " . ($destination - 1)
            . ";\n";
        }
      elsif ($commande eq "JMPC")
        {
          print SMV "    pc = " . ($saut-1) . " & acc\t : " . ($destination - 1)
            . ";\n";
        }
      elsif ($commande eq "JMPCN")
        {
          print SMV "    pc = " . ($saut-1) . " & ~acc\t : " . ($destination - 1)
            . ";\n";
        }
      elsif ($commande eq "RET")
        {
          printf SMV "    pc = " . ($saut-1) . "\t\t : " . ($pc) . ";\n";
        }
      elsif ($commande eq "RETC")
        {
          printf SMV "    pc = " . ($saut-1) . " & acc\t : " . ($pc) . ";\n";
        }
      elsif ($commande eq "RETCN")
        {
          printf SMV "    pc = " . ($saut-1) . " & ~acc\t : " . ($pc) . ";\n";
        }
      else
        {
          die "La commande (de saut?) $commande n'est pas reconnue.\n";
        }
    }
    print SMV "    pc = $pc\t\t : 1;\n";
    print SMV "    default\t\t : pc + 1;\n};\n\n";

    # Lock

    #  print SMV "next(Lock) := case\n";
  #  print SMV "    pc = $pc & !Lock\t : {0,1};\n";
  #  print SMV "    1\t\t : Lock;\n";
  #  print SMV "esac;\n\n";

    # Variables

    foreach $type_var (@types_variables) {
      if ($decl_var{$type_var}) {

        my (@io_type) = split (/ /, $decl_var{$type_var});
```

```
      foreach $variable (@io_type) {
        print SMV "  next($variable) := case\n{\n";

        if ($type_var eq $types_variables[0]) {
          # C'est une variable d'entrée
          print SMV "    pc = $pc & Lock\t : $variable;\n";
          print SMV "    pc = $pc &!Lock\t : {0, 1};\n";
        } else {
          my (@stores) = split (/ /, $store{$variable});
          foreach $l_store (@stores) {
            print SMV "    pc = " . ($l_store - 1) . "\t : acc;\n";
          }
          my (@sets) = split (/ /, $set{$variable});
          foreach $l_set (@sets) {
            print SMV "    pc = " . ($l_set - 1) . "\t : 1;\n";
          }
          my (@resets) = split (/ /, $reset{$variable});
          foreach $l_reset (@resets) {
            print SMV "    pc = " . ($l_reset - 1) . "\t : 0;\n";
          }

        }
        print SMV "    default\t : $variable;\n};\n\n";
      }
    }
  }
}


sub ecrit_formules ()
  {
    print SMV "  attentif : assert G F FDC;\n";

    print SMV "\n}\n";
}
```

D. Vérification du mixeur avec SMV

D.1. Code SMV du programme IL du mixeur

```
#define FDC (pc = 85)
#define TRUE 1
#define FALSE 0

module main ()
{
/* Variables : entree */
  ST : boolean;
  BALANCE_VIDE : boolean;
  POIDS_A : boolean;
  POIDS_AB : boolean;
  MIX_TEMPS : boolean;
  d : boolean;
  S0 : boolean;
  S1 : boolean;
/* Variables : sortie */
  DONE : boolean;
  VA : boolean;
  VB : boolean;
  VC : boolean;
  MT : boolean;
  MR : boolean;
  MP1 : boolean;
  MP0 : boolean;
/* Variables : local */
  START : boolean;
  WEIGH_A : boolean;
  WEIGH_B : boolean;
  FILL : boolean;
  BRICK1 : boolean;
  DROP_1 : boolean;
  BRICK2 : boolean;
  DROP_2 : boolean;
  MIX : boolean;
  TIP : boolean;
  RAISE : boolean;

# Variables internes
```

```
  pc : 1 .. 85;
  acc : boolean;
  Lock : boolean;

  init (pc) := 1;
  init (DONE) := 1;
  init (VA) := 0;
  init (VB) := 0;
  init (VC) := 0;
  init (MT) := 0;
  init (MR) := 0;
  init (MP1) := 0;
  init (MP0) := 0;
  init (START) := 1;
  init (WEIGH_A) := 0;
  init (WEIGH_B) := 0;
  init (FILL) := 0;
  init (BRICK1) := 0;
  init (DROP_1) := 0;
  init (BRICK2) := 0;
  init (DROP_2) := 0;
  init (MIX) := 0;
  init (TIP) := 0;
  init (RAISE) := 0;


  next(acc) := case
{
    pc = 1 : START;
    pc = 3 : ST;
    pc = 4 : acc & S0;
    pc = 5 : acc & BALANCE_VIDE;
    pc = 13 : WEIGH_A;
    pc = 15 : POIDS_A;
    pc = 21 : WEIGH_B;
    pc = 23 : POIDS_AB;
    pc = 30 : BRICK1;
    pc = 32 : d;
    pc = 36 : DROP_1;
    pc = 38 : !d;
    pc = 42 : BRICK2;
    pc = 44 : d;
    pc = 49 : DROP_2;
    pc = 51 : acc & FILL;
    pc = 52 : acc & !d;
    pc = 53 : acc & BALANCE_VIDE;
    pc = 60 : MIX;
    pc = 62 : MIX_TEMPS;
    pc = 67 : TIP;
    pc = 69 : S1;
    pc = 76 : RAISE;
```

```
    pc = 78 : S0;
    default : acc;
};

  next(pc) := case
{
    pc = 2 & ~acc : 13;
    pc = 6 & ~acc : 84;
    pc = 14 & ~acc : 21;
    pc = 16 & ~acc : 30;
    pc = 22 & ~acc : 29;
    pc = 24 & ~acc : 30;
    pc = 31 & ~acc : 36;
    pc = 33 & ~acc : 84;
    pc = 37 & ~acc : 42;
    pc = 39 & ~acc : 84;
    pc = 43 & ~acc : 49;
    pc = 45 & ~acc : 84;
    pc = 50 & ~acc : 60;
    pc = 54 & ~acc : 84;
    pc = 61 & ~acc : 67;
    pc = 63 & ~acc : 84;
    pc = 68 & ~acc : 76;
    pc = 70 & ~acc : 84;
    pc = 77 & ~acc : 84;
    pc = 79 & ~acc : 84;
    pc = 85 : 1;
    default : pc + 1;
};

  next(ST) := case
{
    pc = 85 & Lock : ST;
    pc = 85 & !Lock : {0, 1};
    default : ST;
};

  next(BALANCE_VIDE) := case
{
    pc = 85 & Lock : BALANCE_VIDE;
    pc = 85 & !Lock : {0, 1};
    default : BALANCE_VIDE;
};

  next(POIDS_A) := case
{
    pc = 85 & Lock : POIDS_A;
    pc = 85 & !Lock : {0, 1};
    default : POIDS_A;
};
```

```
  next(POIDS_AB) := case
{
    pc = 85 & Lock : POIDS_AB;
    pc = 85 & !Lock : {0, 1};
    default : POIDS_AB;
};

  next(MIX_TEMPS) := case
{
    pc = 85 & Lock : MIX_TEMPS;
    pc = 85 & !Lock : {0, 1};
    default : MIX_TEMPS;
};

  next(d) := case
{
    pc = 85 & Lock : d;
    pc = 85 & !Lock : {0, 1};
    default : d;
};

  next(S0) := case
{
    pc = 85 & Lock : S0;
    pc = 85 & !Lock : {0, 1};
    default : S0;
};

  next(S1) := case
{
    pc = 85 & Lock : S1;
    pc = 85 & !Lock : {0, 1};
    default : S1;
};

  next(DONE) := case
{
    pc = 82 : 1;
    pc = 7 : 0;
    default : DONE;
};

  next(VA) := case
{
    pc = 10 : 1;
    pc = 17 : 0;
    default : VA;
};

  next(VB) := case
{
```

```
    pc = 19 : 1;
    pc = 25 : 0;
    default : VB;
};

  next(VC) := case
{
    pc = 27 : 1;
    pc = 55 : 0;
    default : VC;
};

  next(MT) := case
{
    pc = 12 : 1;
    pc = 47 : 0;
    default : MT;
};

  next(MR) := case
{
    pc = 59 : 1;
    pc = 73 : 0;
    default : MR;
};

  next(MP1) := case
{
    pc = 65 : 1;
    pc = 72 : 0;
    default : MP1;
};

  next(MP0) := case
{
    pc = 75 : 1;
    pc = 81 : 0;
    default : MP0;
};

  next(START) := case
{
    pc = 83 : 1;
    pc = 8 : 0;
    default : START;
};

  next(WEIGH_A) := case
{
    pc = 9 : 1;
    pc = 18 : 0;
```

```
    default : WEIGH_A;
};

  next(WEIGH_B) := case
{
    pc = 20 : 1;
    pc = 26 : 0;
    default : WEIGH_B;
};

  next(FILL) := case
{
    pc = 28 : 1;
    pc = 56 : 0;
    default : FILL;
};

  next(BRICK1) := case
{
    pc = 11 : 1;
    pc = 34 : 0;
    default : BRICK1;
};

  next(DROP_1) := case
{
    pc = 35 : 1;
    pc = 40 : 0;
    default : DROP_1;
};

  next(BRICK2) := case
{
    pc = 41 : 1;
    pc = 46 : 0;
    default : BRICK2;
};

  next(DROP_2) := case
{
    pc = 48 : 1;
    pc = 57 : 0;
    default : DROP_2;
};

  next(MIX) := case
{
    pc = 58 : 1;
    pc = 64 : 0;
    default : MIX;
};
```

```
  next(TIP) := case
{
    pc = 66 : 1;
    pc = 71 : 0;
    default : TIP;
};

  next(RAISE) := case
{
    pc = 74 : 1;
    pc = 80 : 0;
    default : RAISE;
};

  attentif : assert G F FDC;
/* Étapes mortes */
  etape_start_atteignable : SPEC AG (FDC -> EF (START & FDC));
  etape_weigh_a_atteignable : SPEC AG (FDC -> EF (WEIGH_A & FDC));
  etape_weigh_b_atteignable : SPEC AG (FDC -> EF (WEIGH_B & FDC));
  etape_fill_atteignable : SPEC AG (FDC -> EF (FILL & FDC));
  etape_brick1_atteignable : SPEC AG (FDC -> EF (BRICK1 & FDC));
  etape_drop1_atteignable : SPEC AG (FDC -> EF (DROP_1 & FDC));
  etape_brick2_atteignable : SPEC AG (FDC -> EF (BRICK2 & FDC));
  etape_drop2_atteignable : SPEC AG (FDC -> EF (DROP_2 & FDC));
  etape_mix_atteignable : SPEC AG (FDC -> EF (MIX & FDC));
  etape_tip_atteignable : SPEC AG (FDC -> EF (TIP & FDC));
  etape_raise_atteignable : SPEC AG (FDC -> EF (RAISE & FDC));

/* Lignes mortes */
  ligne_1_atteignable : SPEC AG EF (pc = 1);
  ligne_2_atteignable : SPEC AG EF (pc = 2);
  ligne_3_atteignable : SPEC AG EF (pc = 3);
  ligne_4_atteignable : SPEC AG EF (pc = 4);
  ligne_5_atteignable : SPEC AG EF (pc = 5);
  ligne_6_atteignable : SPEC AG EF (pc = 6);
  ligne_7_atteignable : SPEC AG EF (pc = 7);
  ligne_8_atteignable : SPEC AG EF (pc = 8);
  ligne_9_atteignable : SPEC AG EF (pc = 9);
  ligne_10_atteignable : SPEC AG EF (pc = 10);
  ligne_11_atteignable : SPEC AG EF (pc = 11);
  ligne_12_atteignable : SPEC AG EF (pc = 12);
  ligne_13_atteignable : SPEC AG EF (pc = 13);
  ligne_14_atteignable : SPEC AG EF (pc = 14);
  ligne_15_atteignable : SPEC AG EF (pc = 15);
  ligne_16_atteignable : SPEC AG EF (pc = 16);
  ligne_17_atteignable : SPEC AG EF (pc = 17);
  ligne_18_atteignable : SPEC AG EF (pc = 18);
  ligne_19_atteignable : SPEC AG EF (pc = 19);
  ligne_20_atteignable : SPEC AG EF (pc = 20);
  ligne_21_atteignable : SPEC AG EF (pc = 21);
```

```
ligne_22_atteignable : SPEC AG EF (pc = 22);
ligne_23_atteignable : SPEC AG EF (pc = 23);
ligne_24_atteignable : SPEC AG EF (pc = 24);
ligne_25_atteignable : SPEC AG EF (pc = 25);
ligne_26_atteignable : SPEC AG EF (pc = 26);
ligne_27_atteignable : SPEC AG EF (pc = 27);
ligne_28_atteignable : SPEC AG EF (pc = 28);
ligne_29_atteignable : SPEC AG EF (pc = 29);
ligne_30_atteignable : SPEC AG EF (pc = 30);
ligne_31_atteignable : SPEC AG EF (pc = 31);
ligne_32_atteignable : SPEC AG EF (pc = 32);
ligne_33_atteignable : SPEC AG EF (pc = 33);
ligne_34_atteignable : SPEC AG EF (pc = 34);
ligne_35_atteignable : SPEC AG EF (pc = 35);
ligne_36_atteignable : SPEC AG EF (pc = 36);
ligne_37_atteignable : SPEC AG EF (pc = 37);
ligne_38_atteignable : SPEC AG EF (pc = 38);
ligne_39_atteignable : SPEC AG EF (pc = 39);
ligne_40_atteignable : SPEC AG EF (pc = 40);
ligne_41_atteignable : SPEC AG EF (pc = 41);
ligne_42_atteignable : SPEC AG EF (pc = 42);
ligne_43_atteignable : SPEC AG EF (pc = 43);
ligne_44_atteignable : SPEC AG EF (pc = 44);
ligne_45_atteignable : SPEC AG EF (pc = 45);
ligne_46_atteignable : SPEC AG EF (pc = 46);
ligne_47_atteignable : SPEC AG EF (pc = 47);
ligne_48_atteignable : SPEC AG EF (pc = 48);
ligne_49_atteignable : SPEC AG EF (pc = 49);
ligne_50_atteignable : SPEC AG EF (pc = 50);
ligne_51_atteignable : SPEC AG EF (pc = 51);
ligne_52_atteignable : SPEC AG EF (pc = 52);
ligne_53_atteignable : SPEC AG EF (pc = 53);
ligne_54_atteignable : SPEC AG EF (pc = 54);
ligne_55_atteignable : SPEC AG EF (pc = 55);
ligne_56_atteignable : SPEC AG EF (pc = 56);
ligne_57_atteignable : SPEC AG EF (pc = 57);
ligne_58_atteignable : SPEC AG EF (pc = 58);
ligne_59_atteignable : SPEC AG EF (pc = 59);
ligne_60_atteignable : SPEC AG EF (pc = 60);
ligne_61_atteignable : SPEC AG EF (pc = 61);
ligne_62_atteignable : SPEC AG EF (pc = 62);
ligne_63_atteignable : SPEC AG EF (pc = 63);
ligne_64_atteignable : SPEC AG EF (pc = 64);
ligne_65_atteignable : SPEC AG EF (pc = 65);
ligne_66_atteignable : SPEC AG EF (pc = 66);
ligne_67_atteignable : SPEC AG EF (pc = 67);
ligne_68_atteignable : SPEC AG EF (pc = 68);
ligne_69_atteignable : SPEC AG EF (pc = 69);
ligne_70_atteignable : SPEC AG EF (pc = 70);
ligne_71_atteignable : SPEC AG EF (pc = 71);
ligne_72_atteignable : SPEC AG EF (pc = 72);
```

```
  ligne_73_atteignable : SPEC AG EF (pc = 73);
  ligne_74_atteignable : SPEC AG EF (pc = 74);
  ligne_75_atteignable : SPEC AG EF (pc = 75);
  ligne_76_atteignable : SPEC AG EF (pc = 76);
  ligne_77_atteignable : SPEC AG EF (pc = 77);
  ligne_78_atteignable : SPEC AG EF (pc = 78);
  ligne_79_atteignable : SPEC AG EF (pc = 79);
  ligne_80_atteignable : SPEC AG EF (pc = 80);
  ligne_81_atteignable : SPEC AG EF (pc = 81);
  ligne_82_atteignable : SPEC AG EF (pc = 82);
  ligne_83_atteignable : SPEC AG EF (pc = 83);
  ligne_84_atteignable : SPEC AG EF (pc = 84);
  ligne_85_atteignable : SPEC AG EF (pc = 85);

  /* Exclusions mutuelles */

  exclut_va_vb : assert G (FDC -> ~(VA & VB));
  exclut_va_vc : assert G (FDC -> ~(VA & VC));
  exclut_vb_vc : assert G (FDC -> ~(VB & VC));
  exclut_mp : assert G (FDC -> ~(MP0 & MP1));
  exclut_vpr : assert G (FDC -> ~(MP0 & MR));

  /* Exclusion entre la partie haute et la partie basse */

  exclut_haut_bas : assert G (FDC -> ~((VA | VB | VC | MT) & (MR | MP0 | MP1)));

  /* Les deux phases du haut peuvent se faire en parallèle */

  paral_vannes_mt : SPEC AG (FDC -> EF (((VA | VB | VC) & MT) & FDC));

  /* Les transitions ne sont franchies que si elles doivent l'être */
  start_pas_franchie : assert G (FDC -> ((START & (~BALANCE_VIDE | ~ST | ~S0)) -> X (~FDC U (START & FDC))));

  weigh_a_pas_franchie : assert G (FDC -> ((WEIGH_A & ~POIDS_A) -> X (~FDC U (WEIGH_A & FDC))));

  weigh_b_pas_franchie : assert G (FDC -> ((WEIGH_B & ~POIDS_AB) -> X (~FDC U (WEIGH_B & FDC))));

  brick1_pas_franchie : assert G (FDC -> ((BRICK1 & ~d) -> X (~FDC U (BRICK1 & FDC))));

  tip_pas_franchie : assert G (FDC -> ((TIP & ~S1) -> X (~FDC U (TIP & FDC))));

/* Correspondances entre variables et étapes */

  equiv_start_done : assert G (FDC -> (START <-> DONE));
  equiv_weigh_va : assert G (FDC -> (WEIGH_A <-> VA));
  equiv_weigh_vb : assert G (FDC -> (WEIGH_B <-> VB));
  equiv_fill_vc : assert G (FDC -> (FILL <-> VC));
  equiv_brickdrop_mt : assert G (FDC -> ((BRICK1 | DROP_1 | BRICK2) <-> MT));
}
```

D.2. Rapport de la vérification

```
Copyright 1996 Cadence Berkeley Labs. Cadence Design Systems.


{attentif}
==========
user time.................................................................0.03 s
system time.................................................................0 s

{brick1_pas_franchie}
=====================
user time.................................................................0.05 s
system time.................................................................0 s

{equiv_brickdrop_mt}
====================
user time.................................................................0.06 s
system time.................................................................0 s

{equiv_fill_vc}
===============
user time.................................................................0.07 s
system time..............................................................0.01 s

{equiv_start_done}
==================
user time.................................................................0.09 s
system time..............................................................0.01 s

{equiv_weigh_va}
================
user time.................................................................0.11 s
system time..............................................................0.01 s

{equiv_weigh_vb}
================
user time.................................................................0.12 s
system time..............................................................0.01 s

{exclut_haut_bas}
=================
user time.................................................................0.13 s
```

```
system time...............................................................0.01 s

{exclut_mp}
===========
user time.................................................................0.14 s
system time...............................................................0.01 s

{exclut_va_vb}
==============
user time.................................................................0.14 s
system time...............................................................0.01 s

{exclut_va_vc}
==============
user time.................................................................0.16 s
system time...............................................................0.01 s

{exclut_vb_vc}
==============
user time.................................................................0.17 s
system time...............................................................0.01 s

{exclut_vpr}
============
user time.................................................................0.18 s
system time...............................................................0.01 s

{start_pas_franchie}
====================
user time..................................................................0.2 s
system time...............................................................0.01 s

{tip_pas_franchie}
==================
user time.................................................................0.22 s
system time...............................................................0.01 s

{weigh_a_pas_franchie}
======================
user time.................................................................0.23 s
system time...............................................................0.01 s

{weigh_b_pas_franchie}
```

```
======================
user time..........................................................0.24 s
system time........................................................0.01 s

{etape_start_atteignable}
=========================
user time..........................................................0.25 s
system time........................................................0.01 s

{etape_weigh_a_atteignable}
===========================
user time..........................................................0.26 s
system time........................................................0.01 s

{etape_weigh_b_atteignable}
===========================
user time..........................................................0.27 s
system time........................................................0.01 s

{etape_fill_atteignable}
========================
user time..........................................................0.28 s
system time........................................................0.02 s

{etape_brick1_atteignable}
==========================
user time..........................................................0.29 s
system time........................................................0.02 s

{etape_drop1_atteignable}
=========================
user time...........................................................0.3 s
system time........................................................0.02 s

{etape_brick2_atteignable}
==========================
user time..........................................................0.32 s
system time........................................................0.02 s

{etape_drop2_atteignable}
=========================
user time..........................................................0.33 s
system time........................................................0.02 s
```

```
{etape_mix_atteignable}
=======================
user time.............................................................0.34 s
system time...........................................................0.02 s

{etape_tip_atteignable}
=======================
user time.............................................................0.36 s
system time...........................................................0.02 s

{etape_raise_atteignable}
=========================
user time.............................................................0.37 s
system time...........................................................0.02 s

{ligne_1_atteignable}
=====================
user time.............................................................0.39 s
system time...........................................................0.02 s

{ligne_2_atteignable}
=====================
user time..............................................................0.4 s
system time...........................................................0.02 s

{ligne_3_atteignable}
=====================
user time.............................................................0.41 s
system time...........................................................0.02 s

{ligne_4_atteignable}
=====================
user time.............................................................0.42 s
system time...........................................................0.02 s

{ligne_5_atteignable}
=====================
user time.............................................................0.44 s
system time...........................................................0.02 s

{ligne_6_atteignable}
=====================
```

```
user time.............................................................0.45 s
system time...........................................................0.02 s

{ligne_7_atteignable}
=====================
user time.............................................................0.46 s
system time...........................................................0.02 s

{ligne_8_atteignable}
=====================
user time.............................................................0.48 s
system time...........................................................0.02 s

{ligne_9_atteignable}
=====================
user time.............................................................0.49 s
system time...........................................................0.02 s

{ligne_10_atteignable}
======================
user time..............................................................0.5 s
system time...........................................................0.02 s

{ligne_11_atteignable}
======================
user time.............................................................0.51 s
system time...........................................................0.03 s

{ligne_12_atteignable}
======================
user time.............................................................0.52 s
system time...........................................................0.03 s

{ligne_13_atteignable}
======================
user time.............................................................0.53 s
system time...........................................................0.03 s

{ligne_14_atteignable}
======================
user time.............................................................0.54 s
system time...........................................................0.03 s
```

```
{ligne_15_atteignable}
======================
user time................................................................0.56 s
system time..............................................................0.03 s

{ligne_16_atteignable}
======================
user time................................................................0.58 s
system time..............................................................0.03 s

{ligne_17_atteignable}
======================
user time................................................................0.59 s
system time..............................................................0.03 s

{ligne_18_atteignable}
======================
user time.................................................................0.6 s
system time..............................................................0.03 s

{ligne_19_atteignable}
======================
user time................................................................0.61 s
system time..............................................................0.03 s

{ligne_20_atteignable}
======================
user time................................................................0.62 s
system time..............................................................0.03 s

{ligne_21_atteignable}
======================
user time................................................................0.65 s
system time..............................................................0.03 s

{ligne_22_atteignable}
======================
user time................................................................0.66 s
system time..............................................................0.03 s

{ligne_23_atteignable}
======================
user time................................................................0.68 s
```

```
system time.............................................................0.03 s

{ligne_24_atteignable}
======================
user time...............................................................0.69 s
system time.............................................................0.03 s

{ligne_25_atteignable}
======================
user time...............................................................0.71 s
system time.............................................................0.03 s

{ligne_26_atteignable}
======================
user time...............................................................0.72 s
system time.............................................................0.03 s

{ligne_27_atteignable}
======================
user time...............................................................0.74 s
system time.............................................................0.03 s

{ligne_28_atteignable}
======================
user time...............................................................0.75 s
system time.............................................................0.03 s

{ligne_29_atteignable}
======================
user time...............................................................0.76 s
system time.............................................................0.04 s

{ligne_30_atteignable}
======================
user time...............................................................0.77 s
system time.............................................................0.04 s

{ligne_31_atteignable}
======================
user time...............................................................0.78 s
system time.............................................................0.04 s

{ligne_32_atteignable}
```

```
======================
user time.............................................................0.79 s
system time...........................................................0.04 s

{ligne_33_atteignable}
======================
user time.............................................................0.81 s
system time...........................................................0.04 s

{ligne_34_atteignable}
======================
user time.............................................................0.82 s
system time...........................................................0.04 s

{ligne_35_atteignable}
======================
user time.............................................................0.83 s
system time...........................................................0.04 s

{ligne_36_atteignable}
======================
user time.............................................................0.85 s
system time...........................................................0.04 s

{ligne_37_atteignable}
======================
user time.............................................................0.87 s
system time...........................................................0.04 s

{ligne_38_atteignable}
======================
user time.............................................................0.88 s
system time...........................................................0.04 s

{ligne_39_atteignable}
======================
user time.............................................................0.89 s
system time...........................................................0.04 s

{ligne_40_atteignable}
======================
user time.............................................................0.91 s
system time...........................................................0.04 s
```

```
{ligne_41_atteignable}
======================
user time.............................................................0.92 s
system time...........................................................0.04 s

{ligne_42_atteignable}
======================
user time.............................................................0.93 s
system time...........................................................0.04 s

{ligne_43_atteignable}
======================
user time.............................................................0.95 s
system time...........................................................0.04 s

{ligne_44_atteignable}
======================
user time.............................................................0.97 s
system time...........................................................0.04 s

{ligne_45_atteignable}
======================
user time.............................................................0.98 s
system time...........................................................0.04 s

{ligne_46_atteignable}
======================
user time................................................................1 s
system time...........................................................0.04 s

{ligne_47_atteignable}
======================
user time.............................................................1.01 s
system time...........................................................0.04 s

{ligne_48_atteignable}
======================
user time.............................................................1.03 s
system time...........................................................0.04 s

{ligne_49_atteignable}
======================
```

```
user time..............................................................1.04 s
system time............................................................0.04 s

{ligne_50_atteignable}
======================
user time..............................................................1.06 s
system time............................................................0.04 s

{ligne_51_atteignable}
======================
user time..............................................................1.07 s
system time............................................................0.04 s

{ligne_52_atteignable}
======================
user time..............................................................1.09 s
system time............................................................0.04 s

{ligne_53_atteignable}
======================
user time...............................................................1.1 s
system time............................................................0.04 s

{ligne_54_atteignable}
======================
user time..............................................................1.12 s
system time............................................................0.04 s

{ligne_55_atteignable}
======================
user time..............................................................1.13 s
system time............................................................0.04 s

{ligne_56_atteignable}
======================
user time..............................................................1.14 s
system time............................................................0.04 s

{ligne_57_atteignable}
======================
user time..............................................................1.16 s
system time............................................................0.04 s
```

```
{ligne_58_atteignable}
======================
user time.................................................................1.17 s
system time...............................................................0.04 s

{ligne_59_atteignable}
======================
user time.................................................................1.19 s
system time...............................................................0.04 s

{ligne_60_atteignable}
======================
user time.................................................................1.21 s
system time...............................................................0.04 s

{ligne_61_atteignable}
======================
user time.................................................................1.23 s
system time...............................................................0.04 s

{ligne_62_atteignable}
======================
user time.................................................................1.24 s
system time...............................................................0.04 s

{ligne_63_atteignable}
======================
user time.................................................................1.25 s
system time...............................................................0.04 s

{ligne_64_atteignable}
======================
user time.................................................................1.26 s
system time...............................................................0.05 s

{ligne_65_atteignable}
======================
user time.................................................................1.27 s
system time...............................................................0.05 s

{ligne_66_atteignable}
======================
user time.................................................................1.28 s
```

```
system time.............................................................0.05 s

{ligne_67_atteignable}
======================
user time................................................................1.3 s
system time.............................................................0.05 s

{ligne_68_atteignable}
======================
user time...............................................................1.31 s
system time.............................................................0.05 s

{ligne_69_atteignable}
======================
user time...............................................................1.34 s
system time.............................................................0.05 s

{ligne_70_atteignable}
======================
user time...............................................................1.35 s
system time.............................................................0.05 s

{ligne_71_atteignable}
======================
user time...............................................................1.36 s
system time.............................................................0.05 s

{ligne_72_atteignable}
======================
user time...............................................................1.38 s
system time.............................................................0.05 s

{ligne_73_atteignable}
======================
user time...............................................................1.39 s
system time.............................................................0.05 s

{ligne_74_atteignable}
======================
user time................................................................1.4 s
system time.............................................................0.05 s

{ligne_75_atteignable}
```

```
======================
user time...............................................................1.42 s
system time.............................................................0.05 s

{ligne_76_atteignable}
======================
user time...............................................................1.44 s
system time.............................................................0.05 s

{ligne_77_atteignable}
======================
user time...............................................................1.45 s
system time.............................................................0.05 s

{ligne_78_atteignable}
======================
user time...............................................................1.46 s
system time.............................................................0.05 s

{ligne_79_atteignable}
======================
user time...............................................................1.48 s
system time.............................................................0.05 s

{ligne_80_atteignable}
======================
user time...............................................................1.51 s
system time.............................................................0.05 s

{ligne_81_atteignable}
======================
user time...............................................................1.52 s
system time.............................................................0.05 s

{ligne_82_atteignable}
======================
user time...............................................................1.53 s
system time.............................................................0.05 s

{ligne_83_atteignable}
======================
user time...............................................................1.55 s
system time.............................................................0.05 s
```

```
{ligne_84_atteignable}
======================
user time..............................................................1.56 s
system time............................................................0.05 s

{ligne_85_atteignable}
======================
user time..............................................................1.58 s
system time............................................................0.05 s

{paral_vannes_mt}
=================
user time...............................................................1.6 s
system time............................................................0.05 s

Model checking results
======================
attentif...............................................................true
brick1_pas_franchie....................................................true
equiv_brickdrop_mt.....................................................true
equiv_fill_vc..........................................................true
equiv_start_done.......................................................true
equiv_weigh_va.........................................................true
equiv_weigh_vb.........................................................true
exclut_haut_bas........................................................true
exclut_mp..............................................................true
exclut_va_vb...........................................................true
exclut_va_vc...........................................................true
exclut_vb_vc...........................................................true
exclut_vpr.............................................................true
start_pas_franchie.....................................................true
tip_pas_franchie.......................................................true
weigh_a_pas_franchie...................................................true
weigh_b_pas_franchie...................................................true
etape_start_atteignable................................................true
etape_weigh_a_atteignable..............................................true
etape_weigh_b_atteignable..............................................true
etape_fill_atteignable.................................................true
etape_brick1_atteignable...............................................true
etape_drop1_atteignable................................................true
etape_brick2_atteignable...............................................true
etape_drop2_atteignable................................................true
```

```
etape_mix_atteignable.....................................................true
etape_tip_atteignable.....................................................true
etape_raise_atteignable...................................................true
ligne_1_atteignable.......................................................true
ligne_2_atteignable.......................................................true
ligne_3_atteignable.......................................................true
ligne_4_atteignable.......................................................true
ligne_5_atteignable.......................................................true
ligne_6_atteignable.......................................................true
ligne_7_atteignable.......................................................true
ligne_8_atteignable.......................................................true
ligne_9_atteignable.......................................................true
ligne_10_atteignable......................................................true
ligne_11_atteignable......................................................true
ligne_12_atteignable......................................................true
ligne_13_atteignable......................................................true
ligne_14_atteignable......................................................true
ligne_15_atteignable......................................................true
ligne_16_atteignable......................................................true
ligne_17_atteignable......................................................true
ligne_18_atteignable......................................................true
ligne_19_atteignable......................................................true
ligne_20_atteignable......................................................true
ligne_21_atteignable......................................................true
ligne_22_atteignable......................................................true
ligne_23_atteignable......................................................true
ligne_24_atteignable......................................................true
ligne_25_atteignable......................................................true
ligne_26_atteignable......................................................true
ligne_27_atteignable......................................................true
ligne_28_atteignable......................................................true
ligne_29_atteignable......................................................true
ligne_30_atteignable......................................................true
ligne_31_atteignable......................................................true
ligne_32_atteignable......................................................true
ligne_33_atteignable......................................................true
ligne_34_atteignable......................................................true
ligne_35_atteignable......................................................true
ligne_36_atteignable......................................................true
ligne_37_atteignable......................................................true
ligne_38_atteignable......................................................true
ligne_39_atteignable......................................................true
ligne_40_atteignable......................................................true
```

```
ligne_41_atteignable.......................................................true
ligne_42_atteignable.......................................................true
ligne_43_atteignable.......................................................true
ligne_44_atteignable.......................................................true
ligne_45_atteignable.......................................................true
ligne_46_atteignable.......................................................true
ligne_47_atteignable.......................................................true
ligne_48_atteignable.......................................................true
ligne_49_atteignable.......................................................true
ligne_50_atteignable.......................................................true
ligne_51_atteignable.......................................................true
ligne_52_atteignable.......................................................true
ligne_53_atteignable.......................................................true
ligne_54_atteignable.......................................................true
ligne_55_atteignable.......................................................true
ligne_56_atteignable.......................................................true
ligne_57_atteignable.......................................................true
ligne_58_atteignable.......................................................true
ligne_59_atteignable.......................................................true
ligne_60_atteignable.......................................................true
ligne_61_atteignable.......................................................true
ligne_62_atteignable.......................................................true
ligne_63_atteignable.......................................................true
ligne_64_atteignable.......................................................true
ligne_65_atteignable.......................................................true
ligne_66_atteignable.......................................................true
ligne_67_atteignable.......................................................true
ligne_68_atteignable.......................................................true
ligne_69_atteignable.......................................................true
ligne_70_atteignable.......................................................true
ligne_71_atteignable.......................................................true
ligne_72_atteignable.......................................................true
ligne_73_atteignable.......................................................true
ligne_74_atteignable.......................................................true
ligne_75_atteignable.......................................................true
ligne_76_atteignable.......................................................true
ligne_77_atteignable.......................................................true
ligne_78_atteignable.......................................................true
ligne_79_atteignable.......................................................true
ligne_80_atteignable.......................................................true
ligne_81_atteignable.......................................................true
ligne_82_atteignable.......................................................true
ligne_83_atteignable.......................................................true
```

```
ligne_84_atteignable.......................................................true
ligne_85_atteignable.......................................................true
paral_vannes_mt............................................................true

See file "mix.il0.warn" for warnings.

user time................................................................1.6 s
system time.............................................................0.05 s

Resources used
==============
user time................................................................1.6 s
system time.............................................................0.05 s
BDD nodes allocated..........................................................0
data segment size............................................................0
```

E. Code SMV du programme IL implémentant la commande du chariot

```
#define fdc (pc = 42)
typedef pos 0..31 ;

module main(position,requete,go,stop,ann_req,
            MG,MD,VMC)
{
    input position, requete : pos ;
    input go,stop,ann_req : boolean ;
    output MG,MD,VMC : boolean ;
    mouvement, mission_en_cours : boolean ;
    destination : pos ;

    pc : 0..43 ;
    acc_b : boolean ;
    acc_i : pos ;

    init (pc) := 0 ;
    init (mouvement) := 0 ;
    init (mission_en_cours) := 0 ;
    init (MG) := 0 ;
    init (MD) := 0 ;
    init (VMC) := 0 ;

    next(acc_b) := switch (pc)
        {
            0   : stop ;
            3   : acc_i = position ;
            4   : acc_b & mouvement ;
            6   : ~mouvement ;
```

```
            7   : acc_b & ~go ;
            8   : acc_b & ann_req ;
            10  : ~mouvement ;
            11  : acc_b & go ;
            12  : acc_b & ~ann_req ;
            14  : ~mission_en_cours ;
            16  : 1 ;
            22  : acc_i = destination ;
            24  : 1 ;
            27  : acc_i < destination ;
            29  : 1 ;
            32  : 1 ;
            35  : 0 ;
            38  : 0 ;
            default : acc_b ;
        } ;

    next(acc_i) := switch (pc)
        {
            2   : destination ;
            19  : requete ;
            21  : position ;
            26  : position ;
            default : acc_i ;
        } ;

    next(pc) := case
    {
        pc = 1 & acc_b  : 38 ;
        pc = 5 & acc_b  : 35 ;
        pc = 9 & acc_b  : 35 ;
        pc = 13 & ~acc_b        : 42 ;
        pc = 15 & ~acc_b        : 21 ;
        pc = 23 & acc_b : 42 ;
        pc = 28 & acc_b : 32 ;
        pc = 31         : 42 ;
        pc = 34         : 42 ;
        default         : pc + 1 mod 43 ;
    } ;


    next (go) := switch (pc)
        {
```

```
        0..41 : go ;
        43 : go ;
    } ;

next (stop) := switch (pc)
    {
        0..41 : stop ;
        43 : stop ;
    } ;

next (ann_req) := switch (pc)
    {
        0..41 : ann_req ;
        43 : ann_req ;
    } ;

next (position) := switch (pc)
    {
        0..41 : position ;
        43 : position ;
    } ;

next (requete) := switch (pc)
    {
        0..41 : requete ;
        43 : requete ;
    } ;

next (MG) := switch (pc)
    {
        30  : acc_b ;
        39  : acc_b ;
        default     : MG ;
    } ;

next (MD) := switch (pc)
    {
        33  : acc_b ;
        40  : acc_b ;
        default     : MD ;
    } ;

next (VMC) := switch (pc)
```

```
    {
        18  : acc_b ;
        37  : acc_b ;
        default     : VMC ;
    } ;

next (mouvement) := switch (pc)
    {
        25  : acc_b ;
        41  : acc_b ;
        default     : mouvement ;
    } ;

next (mission_en_cours) := switch (pc)
    {
        17  : acc_b ;
        36  : acc_b ;
        default     : mission_en_cours ;
    } ;

next (destination) := switch (pc)
    {
        20  : acc_i ;
        default     : destination ;
    } ;

moteur_calme : assert G ~(fdc & MG & MD) ;
moteur_calme_interne : assert G ~(MG & MD) ;
enchainement_MG_MD : assert G ((fdc & MG) -> ~ X (~fdc U (fdc & MD))) ;
arret_assure_interne : assert G (stop -> ~mouvement) ;
arret_assure_externe : assert G ((fdc & stop) -> ~mouvement) ;
arret_MG_MD : assert G ((fdc & stop) -> (~MG & ~MD)) ;
arret_moteur_F : assert G (stop -> F (~MG & ~MD)) ;
MG_pas_MD_apres : assert G ((fdc & MG)
                            -> ((G (~fdc | ~MD))
                                | ((~fdc | ~MD) U (fdc & ~MG & ~MD)))) ;
MG_pas_MD_interne : assert G (MG ->
                              ((G ~MD)
                               | (~MD U (~MG & ~MD)))) ;
MD_pas_MG_apres : assert G ((fdc & MD)
                            -> ((G (~fdc | ~MG))
                                | ((~fdc | ~MG) U (fdc & ~MD & ~MG)))) ;
MD_pas_MG_interne : assert G (MD ->
```

```
                                  ((G ~MG)
                                   | (~MG U (~MD & ~MG))));
    go_demarre : assert G ((go & ~ann_req & fdc)-> ((G (~fdc | MG | MD))
                                         | ((~fdc | MG | MD)
                                           U (fdc & (stop
                                                     | (position = destination))))));
    go_demarre_int : assert G ((go & ~ann_req)-> ((G (MG | MD))
                                         | ((MG | MD)
                                           U (stop
                                              | (position = destination)))));
    go_until : assert G ((go & ~ann_req & fdc)-> ((~fdc | MG | MD)
                                                 U (fdc & (stop
                                                           | (position
                                                              = destination)))));
    arrivee : assert G ((MG | MD) -> F (stop | (position = destination)));
    assume arrivee;
    using arrivee prove go_until;
}
```

F. Code SMV du programme ST implémentant la commande du chariot

```
typedef Signal {0, 1, N};
typedef Appel {1, N};
typedef Continuation {1, Exit, Return, N};
typedef entier 0..3;
#define fdc (fin ~= N)


module expr_variable (i, var, o)
{
  input i : Appel;
  output o : Signal;

  init (o) := N;
  next (o) := case
    {
      i = 1 & var = 1 : 1;
      i = 1 & var = 0 : 0;
      i = N : N;
    };
}

module expr_vrai (i, o)
{
  input i : Appel;
  output o : Signal;

  init (o) := N;
  next (o) := case
    {
      i = N : N;
      i = 1 : 1;
    };
}

module expr_faux (i, o)
{
```

```
  input i : Appel ;
  output o : Signal ;

  init (o) := N ;
  next (o) := case
    {
      i = N : N ;
      i = 1 : 0 ;
    } ;
}

module expr_negation (i, ie, oe, o)
{
  input i : Appel ;
  output ie : Appel ;
  input oe : Signal ;
  output o : Signal ;

  init (ie) := N ;
  init (o) := N ;

  next (ie) := i ;
  next (o) := case
    {
      oe = N : N ;
      oe = 1 : 0 ;
      oe = 0 : 1 ;
    } ;
}

module expr_conjonction (i, i1, o1, i2, o2, o)
{
  input i : Appel ;
  output i1, i2 : Appel ;
  input o1, o2 : Signal ;
  output o : Signal ;

  init (i1) := N ;
  init (i2) := N ;
  init (o) := N ;

  next (i1) := i ;
  next (i2) := case
    {
      o1 = 0 : N ;
      o1 = 1 : 1 ;
      default : N ;
    } ;
  next (o) := case
    {
      o1 = 0 : 0 ;
```

```
      default : o2 ;
    } ;
}

module expr_disjonction (i, i1, o1, i2, o2, o)
{
  input i : Appel ;
  output i1, i2 : Appel ;
  input o1, o2 : Signal ;
  output o : Signal ;

  init (i1) := N ;
  init (i2) := N ;
  init (o) := N ;

  next (i1) := i ;
  next (i2) := case
    {
      o1 = 1 : N ;
      o1 = 0 : 1 ;
      default : N ;
    } ;
  next (o) := case
    {
      o1 = 1 : 1 ;
      default : o2 ;
    } ;
}

module expr_egalite (i, i1, o1, i2, o2, o)
{
  input i : Appel ;
  output i1, i2 : Appel ;
  input o1, o2 : Signal ;
  output o : Signal ;
  valeur1 : Signal ;

  init (i1) := N ;
  init (i2) := N ;
  init (o) := N ;
  init (valeur1) := N ;

  next (i1) := i ;
  next (i2) := case
    {
      o1 = 1 : 1 ;
      o1 = 0 : 1 ;
      o1 = N : N ;
    } ;
  next (valeur1) := case
    {
```

```
      o2 ~= N : N ;
      o1 = 1 : 1 ;
      o1 = 0 : 0 ;
      default : valeur1 ;
    } ;
  next (o) := case
    {
      valeur1 = N | o2 = N : N ;
      valeur1 = o2 : 1 ;
      valeur1 ~= o2 : 0 ;
    } ;
}

module expr_egalite_var_entieres (i, var1, var2, o)
{
  input i : Appel ;
  output o : Signal ;
  input var1, var2 : entier ;

  init (o) := N ;
  next (o) := case
    {
      i = 1 & var1 = var2 : 1 ;
      i = 1 & var1 ~=var2 : 0 ;
      i = N : N ;
    } ;
}

module expr_inf_var_entieres (i, var1, var2, o)
{
  input i : Appel ;
  output o : Signal ;
  input var1, var2 : entier ;

  init (o) := N ;
  next (o) := case
    {
      i = 1 & var1 < var2 : 1 ;
      i = 1 & ~(var1 < var2) : 0 ;
      i = N : N ;
    } ;
}

module enonce_affectation (i, ie, oe, o)
{
  input i : Appel ;
  output ie : Appel ;
  input oe : Signal ;
  output o : Continuation ;

  init (ie) := N ;
```

```
  init (o) := N ;

  next (ie) := i ;
  next (o) := case
    {
      oe = 1 : 1 ;
      oe = 0 : 1 ;
      default : N ;
    } ;
}

module enonce_tau (i, o)
{
  input i : Appel ;
  output o : Continuation ;

  init (o) := N ;
  next (o) := case
    {
      i = 1 : 1 ;
      i = N : N ;
    } ;
}

module enonce_exit (i, o)
{
  input i : Appel ;
  output o : Continuation ;

  init (o) := N ;
  next (o) := case
    {
      i = 1 : Exit ;
      i = N : N ;
    } ;
}

module enonce_return (i, o)
{
  input i : Appel ;
  output o : Continuation ;

  init (o) := N ;
  next (o) := case
    {
      i = 1 : Return ;
      i = N : N ;
    } ;
}

module enonce_sequence (i, i1, o1, i2, o2, o)
```

```
{
  input i : Appel ;
  output i1, i2 : Appel ;
  input o1, o2 : Continuation ;
  output o : Continuation ;

  init (i1) := N ;
  init (i2) := N ;
  init (o) := N ;

  next (i1) := i ;
  next (i2) := case
    {
      o1 = 1 : 1 ;
      o1 = Exit : N ;
      o1 = Return : N ;
      default : N ;
    } ;
  next (o) := case
    {
      o1 = Exit | o2 = Exit : Exit ;
      o1 = Return | o2 = Return : Return ;
      o2 = 1 : 1 ;
      default : N ;
    } ;
}

module enonce_selection (i, ie, oe, i1, o1, i2, o2, o)
{
  input i : Appel ;
  output i1, i2, ie : Appel ;
  input oe : Signal ;
  input o1, o2 : Continuation ;
  output o : Continuation ;

  init (ie) := N ;
  init (i1) := N ;
  init (i2) := N ;
  init (o) := N ;

  next (ie) := i ;
  next (i1) := case
    {
      oe = 1 : 1 ;
      default : N ;
    } ;
  next (i2) := case
    {
      oe = 0 : 1 ;
      default : N ;
    } ;
```

```
  next (o) := case
    {
      o1 = Exit | o2 = Exit : Exit ;
      o1 = Return | o2 = Return : Return ;
      o1 = 1 | o2 = 1 : 1 ;
      default : N ;
    } ;
}

module enonce_iteration (i, ie, oe, is, os, o)
{
  input i : Appel ;
  output ie, is : Appel ;
  input oe : Signal ;
  input os : Continuation ;
  output o : Continuation ;

  init (ie) := N ;
  init (is) := N ;
  init (o) := N ;

  next (ie) := case
    {
      i = 1 : 1 ;
      os = 1 : 1 ;
      default : N ;
    } ;
  next (is) := case
    {
      oe = 1 : 1 ;
      default : N ;
    } ;
  next (o) := case
    {
      oe = 0 : 1 ;
      os = Exit : 1 ;
      os = Return : Return ;
      default : N ;
    } ;
}

module main (stop, position, go, ann_req, requete,
             destination, mouvement, MG, MD, mission_en_cours, VMC)
{
  stop, go, ann_req, requete : boolean ;
  position : entier ;
  destination : entier ;
  mouvement, MG, MD, mission_en_cours, VMC : boolean ;

  depart : Appel ;
  fin : Continuation ;
```

```
faux1_i, faux2_i, faux3_i, faux4_i, faux5_i : Appel ;
faux1_o, faux2_o, faux3_o, faux4_o, faux5_o : Signal ;
vrai1_i, vrai2_i, vrai3_i, vrai4_i, vrai5_i : Appel ;
vrai1_o, vrai2_o, vrai3_o, vrai4_o, vrai5_o : Signal ;
var_stop1_i, var_mouvement1_i : Appel ;
var_mouvement2_i, var_mouvement3_i, var_go1_i : Appel ;
var_stop1_o, var_mouvement1_o : Signal ;
var_mouvement2_o, var_mouvement3_o, var_go1_o : Signal ;
var_ann_req1_i, var_go2_i, var_ann_req2_i, var_mission_en_cours1_i : Appel ;
var_ann_req1_o, var_go2_o, var_ann_req2_o, var_mission_en_cours1_o : Signal ;
var_requete1_i : Appel ;
var_requete1_o : Signal ;
expr_negation1_i, expr_negation2_i, expr_negation3_i : Appel ;
expr_negation1_o, expr_negation2_o, expr_negation3_o : Signal ;
expr_negation4_i, expr_negation5_i, expr_negation6_i : Appel ;
expr_negation4_o, expr_negation5_o, expr_negation6_o : Signal ;
expr_conjonction1_i, expr_conjonction2_i, expr_conjonction3_i : Appel ;
expr_conjonction1_o, expr_conjonction2_o, expr_conjonction3_o : Signal ;
expr_conjonction4_i, expr_conjonction5_i : Appel ;
expr_conjonction4_o, expr_conjonction5_o : Signal ;
expr_disjonction1_i : Appel ;
expr_disjonction1_o : Signal ;
expr_egalite1_i, expr_egalite2_i : Appel ;
expr_egalite1_o, expr_egalite2_o : Signal ;
expr_ineg1_i  : Appel ;
expr_ineg1_o : Signal ;

tau1_i, tau2_i, tau3_i, tau4_i, tau5_i : Appel ;
tau1_o, tau2_o, tau3_o, tau4_o, tau5_o : Continuation ;
return1_i, return2_i, return3_i : Appel ;
return1_o, return2_o, return3_o : Continuation ;
aff1_i, aff2_i, aff3_i, aff4_i, aff5_i, aff6_i, aff7_i, aff8_i : Appel ;
aff1_o, aff2_o, aff3_o, aff4_o, aff5_o, aff6_o, aff7_o, aff8_o : Continuation ;
aff9_i, aff10_i, aff11_i : Appel ;
aff9_o, aff10_o, aff11_o : Continuation ;
seq1_i, seq2_i, seq4_i, seq5_i, seq6_i, seq7_i, seq8_i, seq9_i : Appel ;
seq1_o, seq2_o, seq4_o, seq5_o, seq6_o, seq7_o, seq8_o, seq9_o : Continuation ;
if1_i, if2_i, if3_i, if4_i, if5_i, if6_i : Appel ;
if1_o, if2_o, if3_o, if4_o, if5_o, if6_o : Continuation ;

var_stop1 : expr_variable (var_stop1_i, stop, var_stop1_o) ;
var_mouvement1 : expr_variable (var_mouvement1_i, mouvement,
                                var_mouvement1_o) ;
var_mouvement2 : expr_variable (var_mouvement2_i, mouvement,
                                var_mouvement2_o) ;
var_mouvement3 : expr_variable (var_mouvement3_i, mouvement,
                                var_mouvement3_o) ;
var_go1 : expr_variable (var_go1_i, go,
                         var_go1_o) ;
var_go2 : expr_variable (var_go2_i, go,
                         var_go2_o) ;
```

```
var_ann_req1 : expr_variable (var_ann_req1_i, ann_req,
                              var_ann_req1_o);
var_ann_req2 : expr_variable (var_ann_req2_i, ann_req,
                              var_ann_req2_o);
var_mission_en_cours1 : expr_variable (var_mission_en_cours1_i,
                                       mission_en_cours,
                                       var_mission_en_cours1_o);
var_requete1 : expr_variable (var_requete1_i,
                              requete,
                              var_requete1_o);
vrai1 : expr_vrai (vrai1_i, vrai1_o);
vrai2 : expr_vrai (vrai2_i, vrai2_o);
vrai3 : expr_vrai (vrai3_i, vrai3_o);
vrai4 : expr_vrai (vrai4_i, vrai4_o);
vrai5 : expr_vrai (vrai5_i, vrai5_o);
faux1 : expr_faux (faux1_i, faux1_o);
faux2 : expr_faux (faux2_i, faux2_o);
faux3 : expr_faux (faux3_i, faux3_o);
faux4 : expr_faux (faux4_i, faux4_o);
faux5 : expr_faux (faux5_i, faux5_o);
expr_negation1 : expr_negation (expr_negation1_i,
                                var_stop1_i, var_stop1_o,
                                expr_negation1_o);
expr_negation2 : expr_negation (expr_negation2_i,
                                var_mouvement2_i, var_mouvement2_o,
                                expr_negation2_o);
expr_negation3 : expr_negation (expr_negation3_i,
                                var_go1_i, var_go1_o,
                                expr_negation3_o);
expr_negation4 : expr_negation (expr_negation4_i,
                                var_mouvement3_i, var_mouvement3_o,
                                expr_negation4_o);
expr_negation5 : expr_negation (expr_negation5_i,
                                var_ann_req2_i, var_ann_req2_o,
                                expr_negation5_o);
expr_negation6 : expr_negation (expr_negation6_i,
                                var_mission_en_cours1_i,
                                var_mission_en_cours1_o,
                                expr_negation6_o);
expr_conjonction1 : expr_conjonction (expr_conjonction1_i,
                                      var_mouvement1_i, var_mouvement1_o,
                                      expr_egalite1_i, expr_egalite1_o,
                                      expr_conjonction1_o);
expr_conjonction2 : expr_conjonction (expr_conjonction2_i,
                                      expr_negation2_i, expr_negation2_o,
                                      expr_negation3_i, expr_negation3_o,
                                      expr_conjonction2_o);
expr_conjonction3 : expr_conjonction (expr_conjonction3_i,
                                      expr_conjonction2_i, expr_conjonction2_o,
                                      var_ann_req1_i, var_ann_req1_o,
                                      expr_conjonction3_o);
```

```
expr_conjonction4 : expr_conjonction (expr_conjonction4_i,
                                      expr_negation4_i, expr_negation4_o,
                                      var_go2_i, var_go2_o,
                                      expr_conjonction4_o);
expr_conjonction5 : expr_conjonction (expr_conjonction5_i,
                                      expr_conjonction4_i, expr_conjonction4_o,
                                      expr_negation5_i, expr_negation5_o,
                                      expr_conjonction5_o);
expr_disjonction1 : expr_disjonction (expr_disjonction1_i,
                                      expr_conjonction1_i, expr_conjonction1_o,
                                      expr_conjonction3_i, expr_conjonction3_o,
                                      expr_disjonction1_o);
expr_egalite1 : expr_egalite_var_entieres (expr_egalite1_i,
                              destination, position,
                              expr_egalite1_o);
expr_egalite2 : expr_egalite_var_entieres (expr_egalite2_i,
                              destination, position,
                              expr_egalite2_o);
expr_ineg1 : expr_inf_var_entieres (expr_ineg1_i, position, destination,
                                    expr_ineg1_o);

tau1 : enonce_tau (tau1_i, tau1_o);
tau2 : enonce_tau (tau2_i, tau2_o);
tau3 : enonce_tau (tau3_i, tau3_o);
tau4 : enonce_tau (tau4_i, tau4_o);
tau5 : enonce_tau (tau5_i, tau5_o);
return1 : enonce_return (return1_i, return1_o);
return2 : enonce_return (return2_i, return2_o);
return3 : enonce_return (return3_i, return3_o);
aff1 : enonce_affectation (aff1_i, faux1_i, faux1_o, aff1_o);
aff2 : enonce_affectation (aff2_i, faux2_i, faux2_o, aff2_o);
aff3 : enonce_affectation (aff3_i, faux3_i, faux3_o, aff3_o);
aff4 : enonce_affectation (aff4_i, faux4_i, faux4_o, aff4_o);
aff5 : enonce_affectation (aff5_i, faux5_i, faux5_o, aff5_o);
aff6 : enonce_affectation (aff6_i, vrai1_i, vrai1_o, aff6_o);
aff7 : enonce_affectation (aff7_i, vrai2_i, vrai2_o, aff7_o);
aff8 : enonce_affectation (aff8_i, var_requete1_i, var_requete1_o, aff8_o);
aff9 : enonce_affectation (aff9_i, vrai3_i, vrai3_o, aff9_o);
aff10 : enonce_affectation (aff10_i, vrai4_i, vrai4_o, aff10_o);
aff11 : enonce_affectation (aff11_i, vrai5_i, vrai5_o, aff11_o);
seq1 : enonce_sequence (seq1_i, aff1_i, aff1_o, aff2_i, aff2_o, seq1_o);
seq2 : enonce_sequence (seq2_i, seq1_i, seq1_o, aff3_i, aff3_o, seq2_o);
seq3 : enonce_sequence (depart, if1_i, if1_o, seq2_i, seq2_o, fin);
seq4 : enonce_sequence (seq4_i, aff4_i, aff4_o, aff5_i, aff5_o, seq4_o);
seq5 : enonce_sequence (seq5_i, if4_i, if4_o, if5_i, if5_o, seq5_o);
seq6 : enonce_sequence (seq6_i, aff6_i, aff6_o, aff7_i, aff7_o, seq6_o);
seq7 : enonce_sequence (seq7_i, seq6_i, seq6_o, aff8_i, aff8_o, seq7_o);
seq8 : enonce_sequence (seq8_i, aff9_i, aff9_o, if6_i, if6_o, seq8_o);
seq9 : enonce_sequence (seq9_i, seq5_i, seq5_o, return3_i, return3_o, seq9_o);
if1 : enonce_selection (if1_i, expr_negation1_i, expr_negation1_o,
                                if2_i, if2_o,
```

```
                               tau1_i, tau1_o,
                        if1_o) ;
if2 : enonce_selection (if2_i, expr_disjonction1_i, expr_disjonction1_o,
                               seq4_i, seq4_o,
                               if3_i, if3_o,
                        if2_o) ;
if3 : enonce_selection (if3_i, expr_conjonction5_i, expr_conjonction5_o,
                               seq9_i, seq9_o,
                               return1_i, return1_o,
                        if3_o) ;
if4 : enonce_selection (if4_i, expr_negation6_i, expr_negation6_o,
                               seq7_i, seq7_o,
                               tau5_i, tau5_o,
                        if4_o) ;
if5 : enonce_selection (if5_i, expr_egalite2_i, expr_egalite2_o,
                               return2_i, return2_o,
                               seq8_i, seq8_o,
                        if5_o) ;
if6 : enonce_selection (if6_i, expr_ineg1_i, expr_ineg1_o,
                        aff10_i, aff10_o,
                        aff11_i, aff11_o,
                        if6_o) ;

/* Libre : tau2, tau3, tau4 */

init (stop) := 0 ;
next (stop) := case
  {
    fin ~= N : {0, 1} ;
    default : stop ;
  } ;

init (position) := 0 ;
next (position) := case
  {
    fin ~= N : {0, 1} ;
    default : position ;
  } ;

init (go) := 0 ;
next (go) := case
  {
    fin ~= N : {0, 1} ;
    default : go ;
  } ;

init (ann_req) := 0 ;
next (ann_req) := case
  {
    fin ~= N : {0, 1} ;
    default : ann_req ;
```

```
    };

  init (requete) := 0;
  next (requete) := case
    {
      fin ~= N : {0, 1};
      default : requete;
    };

  init (destination) := 0;
  next (destination) := case
    {
      aff8.oe ~= N : requete;
      1 : destination;
    };

  init (mouvement) := 0;
  next (mouvement) := case
    {
      aff1.oe = 1 : 1;
      aff1.oe = 0 : 0;
      aff9.oe = 1 : 1;
      aff9.oe = 0 : 0;
      default : mouvement;
    };

  init (MG) := 0;
  next (MG) := case
    {
      aff2.oe = 1 : 1;
      aff2.oe = 0 : 0;
      aff11.oe = 1 : 1;
      aff11.oe = 0 : 0;
      default : MG;
    };

  init (MD) := 0;
  next (MD) := case
    {
      aff3.oe = 1 : 1;
      aff3.oe = 0 : 0;
      aff10.oe = 1 : 1;
      aff10.oe = 0 : 0;
      default : MD;
    };

  init (mission_en_cours) := 0;
  next (mission_en_cours) := case
    {
      aff4.oe = 1 : 1;
      aff4.oe = 0 : 0;
```

```
      aff6.oe = 1 : 1 ;
      aff6.oe = 0 : 0 ;
      default : mission_en_cours ;
    } ;

  init (VMC) := 0 ;
  next (VMC) := case
    {
      aff5.oe = 1 : 1 ;
      aff5.oe = 0 : 0 ;
      aff7.oe = 1 : 1 ;
      aff7.oe = 0 : 0 ;
      default : VMC ;
    } ;

  init (depart) := 1 ;
  next (depart) := case
    {
      fin = 1 : 1 ;
      fin = Return : 1 ;
      fin = Exit : 1 ;
      default : N ;
    } ;

  moteur_calme : assert G ~(fdc & MG = 1 & MD = 1) ;
  moteur_calme_interne : assert G ~(MG = 1 & MD = 1) ;
  enchainement_MG_MD : assert G ((fdc & MG = 1)
                               -> X (~fdc U (fdc & (MD = 0)))) ;
  /* arret_assure_interne : assert G (stop = 1 -> mouvement = 0) ; */
  arret_assure_externe : assert G ((fdc & stop = 1) -> mouvement = 0) ;
  arret_MG_MD : assert G ((fdc & stop = 1) -> (MG = 0 & MD = 0)) ;
  arret_moteur_F : assert G (stop = 1 -> F (MG = 0 & MD = 0)) ;
  MG_pas_MD_apres : assert G ((fdc & MG = 1)
                            -> ((G (~fdc | MD = 0))
                               | ((~fdc | MD = 0) U (fdc & MG = 0 & MD = 0)))) ;
  MG_pas_MD_interne : assert G (MG = 1 ->
                               ((G MD = 0)
                                | (MD = 0 U (MG = 0 & MD = 0)))) ;
  MD_pas_MG_apres : assert G ((fdc & MD = 1)
                             -> ((G (~fdc | MG = 0))
                                | ((~fdc | MG = 0) U (fdc & MD = 0 & MG = 0)))) ;
  MD_pas_MG_interne : assert G (MD = 1 ->
                               ((G MG = 0)
                                | (MG = 0 U (MD = 0 & MG = 0)))) ;
  go_demarre : assert G ((go = 1 & ann_req = 0 & fdc)-> ((G (~fdc | MG = 1 | MD = 1))
                                      | ((~fdc | MG = 1 | MD = 1)
                                         U (fdc & (stop = 1
                                                   | (position = destination)))))) ;
/*  go_demarre_int : assert G ((go = 1 & ann_req = 0)-> ((G (MG = 1 | MD = 1))
                                      | ((MG = 1 | MD = 1)
                                         U (stop = 1
```

```
                                              | (position = destination))))) ; */
  go_until : assert G ((go = 1 & ann_req = 0 & fdc)-> ((~fdc | MG = 1 | MD = 1)
                                                 U (fdc & (stop = 1
                                                           | (position
                                                              = destination))))) ;
  arrivee : assert G ((MG = 1 | MD = 1) -> F (stop = 1 | (position = destination))) ;
  assume arrivee ;
  using arrivee prove go_until ;
}
```