Modélisation et validation des systèmes réactifs : un langage synchrone à base d'automates

Florence Maraninchi

Document d'Habilitation à Diriger des Recherches
Université Joseph Fourier - Grenoble I

Modélisation et validation des systèmes réactifs : un langage synchrone à base d'automates

---

Florence Maraninchi

Soutenance le 22 mai 1997
devant un jury composé de :

Michel Adiba	*président*
André Arnold	*rapporteur*
Gérard Berry	*rapporteur*
Guy Vidal-Naquet	*rapporteur*
Nicolas Halbwachs	*examinateur*
Pierre-Claude Scholl	*examinateur*

Remerciements

Je remercie tout d'abord Michel Adiba pour avoir accepté la présidence du jury ; André Arnold, Gérard Berry et Guy Vidal-Naquet pour avoir accepté de juger mon travail, et pour leurs suggestions de nouvelles pistes ; Nicolas Halbwachs et Pierre-Claude Scholl pour leur participation au jury, et les nombreuses discussions qui ont contribué à la forme et au fond de ce document.

Ensuite j'aimerais profiter de cette page pour citer, plus généralement, toutes les personnes avec qui j'ai travaillé et qui m'ont appris mon métier.

En commençant par le contexte d'enseignement, je tiens à remercier vivement les collègues qui m'ont accueillie en 1990 et ont supporté ma naïveté avec une indulgence et une bonne humeur que le recul me fait apprécier davantage encore. A l'Ensimag, il s'agit de Roger Mohr, Jacques Mossière, Jean-Louis Roch et Yann Rouzaud ; à l'UFR IMA, il s'agit des équipes L&P (Philippe Bizard, Jean-Pierre Peyrin et Catherine Berrut) et MPI (Pierre-Claude Scholl, Marie-Christine Fauvet et Fabienne Lagnier).
Pour la période plus récente, merci de nouveau à l'équipe MPI, pour tout ce que m'ont appris les séances de rédaction de l'ouvrage [SFLM93], et pour l'ambiance de travail en DEUG toujours aussi agréable.
Merci aussi à mes collègues de l'équipe ALM actuelle (Paul Amblard, Jean-Claude Fernandez, Fabienne Lagnier, Pascal Sicard et Philippe Waille), avec qui l'on peut discuter de portes NAND, de compilation, des mystères du fork et de bien d'autres sujets, selon l'humeur du jour. Les séances de rédaction du futur ouvrage "Architectures Logicielles et Matérielles" m'ont déjà beaucoup appris, et ce n'est sans doute pas fini !

Côté recherche, merci à Joseph Sifakis pour m'avoir accueillie dans son équipe dès mon DEA en 87, en me proposant le sujet qui a orienté tout mon travail ultérieur. Sous sa direction VÉRIMAG est un environnement de travail incomparable, plein de bonne humeur et foisonnant.
Nicolas Halbwachs et Gérard Berry m'ont expliqué pourquoi mon travail avait quelque chose à voir avec le domaine de la programmation synchrone, et sont donc responsables de l'apparition d'ARGOS dans la communauté synchrone. Merci à eux et à tous mes collègues du sous-ensemble synchrone de Vérimag, qui savent supporter le rythme de travail irrégulier des enseignants.
Merci aussi à tous les étudiants avec lesquels j'ai travaillé et qui m'ont aidée : Ch. Bard, G. Bodoin, D. Franck, J.-P. Schmidt, T. Popovici, P. Laffont et Y. Rémond. Ph. Schaar et M. Jourdan ont beaucoup fait avancer la définition d'Argos et l'étude de l'environnement de programmation associé.

Avant-propos

Il n'y a pas, à ma connaissance, de règle communément admise sur ce qui peut constituer un document d'habilitation à diriger des recherches. Il est bon, toutefois, de s'interroger sur la finalité d'un tel document, outre la satisfaction personnelle de rassembler et d'organiser des idées jusque là encore un peu dispersées.

Il m'a semblé qu'un exposé purement technique n'était pas de nature à mettre en valeur l'aptitude à organiser, orienter des recherches, sans même parler de les diriger. Il apparaissait donc qu'il fallût plutôt présenter une démarche, un ensemble structuré de motivations, montrer comment les problèmes étaient apparus, comment s'effectuent les choix et comment les travaux sont articulés. Vaste programme...

La difficulté de l'exercice réside dans le fait qu'il est impossible de présenter une démarche indépendamment du sujet technique sur lequel elle s'est exercée, lorsque cette démarche s'est forgée de pair avec le sujet. Le présent document revêt donc un certain caractère autobiographique dans l'exposé de l'approche suivie et de ses évolutions, en particulier lorsqu'il s'agit de reconnaître les influences extérieures qui ont orienté mon travail. Cela me paraît toutefois relever de la plus élémentaire honnêteté intellectuelle et mériter, à ce titre, de figurer ici.

Ce document n'a pas d'ambition de présentation synthétique d'un domaine de recherche, ce qui serait présompteux. Ce n'est pas non plus un document très technique présentant un ensemble de résultats fondamentaux. J'aimerais plutôt y démontrer une certaine cohérence des directions suivies, et comment les résultats convergent. Ce n'est au fond qu'une mise en forme des réflexions que m'inspirent dix années de travail qui, certes, ont conduit à la définition et à l'implantation d'un langage, mais qui, surtout, constituent mon apprentissage du métier d'enseignant/chercheur.

L'essentiel des travaux décrits, toutefois, concernent la définition et l'implantation d'un langage de programmation synchrone baptisé ARGOS, pour la modélisation et la validation des systèmes dits "*réactifs*", et ce document devrait pouvoir constituer, au moins pendant un certain temps, le document de référence sur ARGOS.

Les travaux décrits ici ont été réalisés en parallèle, ou plutôt en étroite relation, avec une activité d'enseignement à plein temps, sur un poste de Maître de Conférences. Il me paraît peu souhaitable, et de toute façon difficile, de tracer une frontière nette entre les activités de recherche et d'enseignement.

La diffusion des résultats ou de la culture d'une équipe de recherche par des interventions dans des formations de troisième cycle — particulièrement en DEA — est naturelle, répandue, et offre en général peu de surprises. Il est en revanche moins courant de pouvoir utiliser cette culture pour enrichir des enseignements de premier ou second cycle, et ceci à court terme. La section MPI (*mathématiques, physique et informatique*) du DEUG scientifique de Grenoble, et l'équipe d'enseignants qui m'y a accueillie dès 1990, m'ont offert un champ d'expérimentation passionnant. J'intègre à ce document la description de deux activités d'enseignement pour lesquelles le lien

avec des activités de recherche est évident. Je mentionne, lorsque cela est le cas, les convergences entre des modèles développés dans les deux environnements de travail, à des fins différentes, mais en partant de la même idée.

Inversement, l'influence des activités d'enseignement sur les activités de recherche ne se limite pas, comme on pourrait le croire, à un manque de disponibilité. Elle prend plusieurs formes. Enseigner les bases de données et le logiciel de base tout en travaillant sur la programmation des systèmes réactifs, peut paraître curieux, mais permet des rapprochements de points de vue qui sinon sembleraient audacieux. On ne connaît jamais si bien les algorithmes sur les graphes et leurs complexités respectives qu'après les avoir décortiqués en compagnie de 30 étudiants. Quant aux réflexions sur le rôle central des langages dans l'apprentissage de l'informatique, menées dans le cadre d'un enseignement de base en premier cycle, elles ne peuvent que profiter à la conception d'un nouveau langage, fût-il dédié aux systèmes réactifs.

D'autre part la nécessaire diversité des sujets abordés en sept années de plein service d'enseignement à des publics variés permet une approche globale d'un sujet de recherche, de la formulation théorique aux problèmes d'implantation.

Enfin, et c'est peut-être le plus important, l'ensemble des collègues que l'on côtoie régulièrement dans un contexte d'enseignement représente une grande diversité de cultures — scientifiques ou non — et, l'enseignant étant souvent bavard, les occasions d'en profiter sont nombreuses. Dans un environnement de recherche, la nécessaire concentration des efforts laisse peu de place aux manifestations de pure curiosité pour des sujets qui ne relèvent pas directement d'un projet.

J'ajouterai que la fragmentation du temps consacré aux activités de recherche, si elle constitue le handicap principal des enseignants dans une équipe de recherche, peut aussi avoir des effets bénéfiques, en obligeant à des prises de recul plus fréquentes, et en détruisant dans l'oeuf toute velléité de ré-inventer la roue.

Table des matières

Chapitre 1

Introduction

1.1 Systèmes réactifs

La typologie désormais consacrée proposée par A. Pnueli [HP88] distingue parmi la grande variété des systèmes informatiques les systèmes *réactifs* et les systèmes *transformationnels*.

Les derniers sont les systèmes classiques de traitement de données, comme les compilateurs. Ils mettent en oeuvre des structures de données et des algorithmes complexes, mais posent en général peu de problèmes d'interaction avec un monde extérieur.

Au contraire, les systèmes réactifs sont caractérisés par la prépondérance des aspects d'interaction avec un environnement, que ce soit un utilisateur humain ou un processus physique. Ces aspects d'interaction sont d'autant plus difficiles à décrire que ces systèmes sont souvent intrinsèquement parallèles. G. Berry a introduit une distinction supplémentaire entre systèmes *interactifs*, pour lesquels le rythme de l'interaction est fixé par le système, et systèmes réactifs, pour lesquels le rythme de l'interaction est fixé par l'environnement. Les systèmes réactifs doivent donc satisfaire des propriétés temporelles imposées par l'environnement, portant par exemple sur le temps de réponse aux stimuli extérieurs. C'est la raison pour laquelle on classe les interfaces homme/machine ou les systèmes d'exploitation parmi les systèmes "seulement" interactifs, et les contrôleurs de processus industriels parmi les systèmes réactifs.

La programmation des systèmes réactifs constitue un domaine de recherche en soi. D'une part, les langages séquentiels classiques sont mal adaptés à la description d'une interaction complexe avec un environnement ; d'autre part ces systèmes sont souvent *critiques*, au sens où leur défaillance peut entraîner de graves conséquences matérielles ou humaines. Un environnement de programmation des systèmes réactifs doit donc proposer des outils et des méthodes de validation formelle. En se restreignant à l'étude des systèmes réactifs, il est possible de développer des langages et des approches de vérification dédiés, en profitant par exemple du fait que certaines abstractions raisonnables des systèmes considérés peuvent être décrites par des objets finis, sur lesquels toute propriété imaginable devient décidable.

Description du noyau réactif d'un système

Pour mettre en évidence les aspects d'interaction propres aux systèmes interactifs ou réactifs, on peut considérer que l'exécution d'un système suit toujours le schéma :

Initialisations
tantque vrai
    Acquérir des entrées e
    Calculer des sorties s
    Emettre les sorties s

La boucle est exécutée un grand nombre de fois, peut-être même indéfiniment : puisque l'on cherche à décrire les aspects d'interaction avec l'environnement, le système a un comportement intéressant même si son exécution ne se termine pas. (Note : on pourrait se ramener à ce schéma pour les systèmes transformationnels. Tout se passe comme si la boucle était exécutée une seule fois, l'expression 'calcul des sorties' représentant tout le travail, éventuellement complexe, de transformation des données en résultats). Cette exécution en boucle est à la base du fonctionnement des automates programmables.

On appelle *noyau réactif* du système le programme chargé de l'étape de calcul des sorties. L'exécution du système est partitionnée en *cycles*. Si l'on imagine une échelle de temps logique discret correspondant aux cycles d'exécution, on peut définir la suite des entrées et la suite des sorties indicées par des entiers naturels[1].

Si le calcul des sorties ne dépend que de l'entrée courante — $s_n = f(e_n)$ —, on obtient un modèle d'exécution assimilable aux circuits combinatoires. En général, toutefois, la sortie au cycle n dépend de l'histoire des entrées : un noyau réactif est un programme qui implante une fonction de la forme : $s_n = f(e_0, e_1, ...e_n)$. La nième sortie dépend de l'histoire des entrées, jusqu'au rang n inclus. Si le système fonctionne indéfiniment, l'histoire des entrées n'est pas bornée.

Systèmes à mémoire bornée

Dans le domaine des systèmes réactifs, on ne s'intéresse qu'à un sous-ensemble de ces systèmes, dits *à mémoire bornée*, pour lesquels une abstraction bornée de l'histoire des entrées suffit à calculer la sortie courante[2]. De manière générale, un noyau réactif fait donc intervenir une variable "*mémoire*" pour stocker l'abstraction de l'histoire courante des entrées, et suit un algorithme de la forme :

Initialisation de la mémoire M
tantque vrai
    Acquérir des entrées e
    $s = f(M, e)$ — *calcul de la sortie courante*
    $M = g(M, e)$ — *mise à jour de la mémoire*
    Emettre les sorties s

où M représente la mémoire des entrées.

Exemple 1.1 : Un système réactif à mémoire booléenne
Imaginons un système qui lit un flot d'entrée composé de caractères a et b uniquement, et qui émet un caractère c sur son flot de sortie, chaque fois qu'il lit un b, à condition qu'il ait lu

1. Certaines implémentations de systèmes réactifs sont telles que le noyau réactif ainsi défini est activé sur interruptions, plutôt qu'exécuté périodiquement. La différence entre les deux approches a peu d'influence sur la suite, l'échelle de temps logique n'étant pas nécessairement régulière en temps physique.

2. La mémoire de tout système informatique est bornée, à l'exécution, pour des raisons physiques ; mais la taille mémoire nécessaire à l'exécution peut ne pas être prévisible. Or un programme réactif ne peut se permettre les arrêts intempestifs pour cause d'échec de l'allocation mémoire. La mémoire doit donc être bornée statiquement.

un nombre pair de a auparavant. Ce système fonctionne avec une mémoire booléenne. En effet, l'abstraction nécessaire et suffisante de l'histoire des entrées est le nombre d'entrées a rencontrées, modulo 2. On a :

$$f(M, e) = \text{ si } M \ \wedge \ e \ni b \text{ alors } c \text{ sinon } \emptyset$$
$$g(M, e) = \text{ si } e \ni a \text{ alors } \neg M \text{ sinon } M$$

(On note $e \ni b$ lorsque le caractère b apparaît dans l'entrée e). Les valeurs de la mémoire M alternent avec les occurrences de a. Le caractère c est émis lorsque b apparaît, à la condition que la mémoire M soit vraie. C'est un choix arbitraire, si l'on ne dit pas comment initialiser la mémoire M. Pour assurer un comportement conforme à la spécification, on doit initialiser M à vrai. □

Une grande partie du travail décrit dans ce document est basée sur les équivalences entre diverses formes de description des noyaux réactifs : a) algorithmes itératifs définis ci-dessus ; b) fonctions à mémoire bornée des séquences d'entrées vers les séquences de sorties ; c) ensembles d'équations de flots ; d) machines de Mealy ; e) circuits booléens séquentiels. Ainsi la programmation multi-langages ARGOS+LUSTRE (chapitre 8) s'appuie-t-elle sur l'équivalence des modèles b) et d) ; la proposition de description des algorithmes itératifs en ARGOS (chapitre 14) sur l'équivalence entre a) et d) ; la compilation symbolique d'ARGOS (chapitre 5) sur l'équivalence entre c) et d) ; la description des circuits en LUSTRE et VHDL sur l'équivalence entre c) et e), etc.

L'autre partie du travail présenté ici porte sur les aspects de définition d'un *langage*, mentionnés plus loin.

1.2 Modèles pour les systèmes réactifs

Tenter d'établir une liste exhaustive des langages et formalismes proposés pour la description, la vérification ou la programmation des systèmes réactifs relève de l'utopie. Nous ne nous intéressons qu'à un sous-ensemble, déjà vaste, de ce domaine, qui comprend les méthodes et outils basés sur la définition d'un *modèle* des systèmes étudiés.

Les *modèles* sont des structures *mathématiques* qui peuvent être plus ou moins riches selon les systèmes étudiés, ou selon le niveau de détail désiré dans la description. Ils peuvent également avoir des représentations *informatiques* différentes selon les outils (génération de code, vérification...) qu'on leur applique. Définir un modèle approprié à la modélisation des systèmes que l'on étudie permet de raisonner formellement sur ces systèmes, par exemple pour réaliser des vérifications.

Donnons tout de suite un exemple. Les systèmes à mémoire bornée peuvent être représentés par des machines à états finies, chaque état correspondant à une valeur possible de la mémoire. Les transitions représentent les changement d'états, c'est-à-dire la mise à jour de la mémoire et les sorties produites, en fonction des entrées lues. Les machines de Mealy (dont les transitions portent des couples entrée/sortie) constituent un modèle naturel des systèmes réactifs. Selon la nature des étiquettes de transitions (atomes, formules booléennes, formules arithmétiques, ...) on obtient des modèles plus ou moins riches.

Les traitements applicables et les questions auxquelles on sait répondre dépendent de la nature des informations associées aux états et transitions. Par exemple, si les transitions portent des formules booléennes, tout problème d'accessibilité à partir d'un état initial donné est décidable. Si les transitions portent des conditions quelconques sur des variables de types quelconques (typiquement des réels), un grand nombre de propriétés deviennent indécidables. Dans les chapitres

qui suivent on rencontrera :

- les systèmes de transitions à étiquettes abstraites (format d'entrée de l'outil ALDÉBARAN, chapitre 13)
- les machines de Mealy booléennes (définition d'ARGOS, chapitre 2), dans lesquelles les étiquettes sont enrichies d'une structure entrée/sortie, la condition d'entrée étant une formule booléenne quelconque.
- les automates interprétés généraux (compilation en C, chapitre 13) dont les transitions portent des conditions sur, et des affectations à, des variables de types quelconques, de telle sorte que le modèle a la puissance d'expression des machines de Turing, et aussi peu de propriétés de décidabilité que lesdites machines.
- les automates temporisés (ou "*Timed Graphs*" [AD90]) utilisés au chapitre 10, dont les transitions portent des conditions sur, et des affectations à, des variables réelles. Les conditions sont des comparaisons à des bornes entières et les affectations sont des remises à zéro. Cette restriction assure la décidabilité des formules de TCTL, une extension de la logique temporelle CTL dans laquelle les modalités sont restreintes par des bornes exprimées de manière quantitative.
- les automates hybrides (chapitre 11), plus expressifs que les automates temporisés, dont les états et les transitions sont étiquetés par des systèmes d'inéquations linéaires, et qu'on ne sait analyser que par des techniques approchées.

De nombreux autres modèles sont étudiés, pour leurs propriétés de décidabilité et d'expressivité. On trouve en particulier toute une hiérarchie de modèles entre les automates temporisés et les automates hybrides.

D'un point de vue informatique, on peut représenter ces machines en extension, en donnant explicitement l'ensemble des états et des transitions, ou symboliquement, par des systèmes d'équations.

1.3 Les langages synchrones

Les modèles (par exemple les machines de Mealy) sont rarement utilisables pour décrire directement les systèmes étudiés. Ils ne peuvent que constituer une représentation intermédiaire entre un véritable *langage*, et les outils de génération de code ou de vérification. Un langage de description des systèmes doit posséder une sémantique formelle qui décrit comment les programmes peuvent être compilés en modèles. Il est courant d'exiger de cette sémantique quelques bonnes propriétés comme la compositionnalité.

Parmi les nombreux formalismes et langages proposés pour décrire des systèmes réactifs, l'approche synchrone est sans doute celle qui fait les hypothèses les plus simples. Les langages synchrones les plus connus sont ESTEREL [BG92], LUSTRE [CHPP87] et SIGNAL [GBBG85], tous trois développés en France. Les Statecharts [Har87] sont parfois rangés parmi les langages synchrones, bien que leur sémantique ne relève pas tout à fait de l'application stricte de l'hypothèse de synchronisme. On trouve également de nombreux formalismes ou langages qui relèvent des hypothèses de la famille synchrone, et pourrait donc y être rangés, bien que leurs auteurs n'utilisent pas le même vocabulaire. C'est le cas en particulier pour le GRAFCET. Les équipes de Ch. André à Nice et de L. Marcé à Brest ont travaillé sur la sémantique synchrone du Grafcet, par

exemple par traduction en ESTEREL. Il est intéressant de noter que ces langages existent, parfois depuis longtemps, non seulement dans le monde universitaire, mais aussi dans l'industrie. C'est d'ailleurs cet heureux "hasard" qui assure le succès du transfert de LUSTRE.

Les langages synchrones proposent des constructions qui permettent de décrire un système comme la composition (par exemple parallèle) de plusieurs sous-systèmes communiquant par diffusion synchrone. L'*hypothèse de synchronisme* se résume à : *le temps de réaction d'un système est nul*, autrement dit les sorties sont considérées comme simultanées aux entrées qui les provoquent. Les arguments sur les avantages de l'approche synchrone, dont la réfutation de la critique usuelle selon laquelle le modèle est complètement irréaliste, sont développés dans [IEE91], et je ne m'y attarderai pas. Notons simplement que l'hypothèse de synchronisme a deux facettes.

Si l'on considère la relation entre deux sous-systèmes d'un même programme, l'hypothèse de réaction en temps nul simplifie considérablement la sémantique de la composition, et permet de définir simplement les opérations du langage — les langages synchrones étant dédiés à l'élaboration de programmes séquentiels, destinés à une exécution centralisée. La composition parallèle n'est donc qu'une facilité de description, elle est *compilée*, et n'a rien à voir avec un parallélisme d'exécution. Paul Caspi et Alain Girault ont étudié le problème de répartition des programmes synchrones [GC92, CGP94].

Si l'on considère la relation entre le système global et l'environnement dans lequel il s'exécute, l'hypothèse permet de s'abstraire, pendant la conception du programme, des paramètres de temps d'exécution liés à la machine. Sur le modèle d'exécution proposé plus haut, la relation entre temps logique du système et temps physique, qui doit être étudiée pour prendre en compte les contraintes de temps des systèmes réactifs, est simple. La phase de calcul des sorties doit être de complexité raisonnable, puisque c'est sur elle que portent les contraintes de temps : si le rythme de production des entrées par l'environnement est de une toutes les millisecondes, le programme doit pouvoir faire un tour dans la boucle en moins d'une milliseconde, pour ne pas risquer d'ignorer des entrées.

ESTEREL et LUSTRE sont compilables en automates interprétés généraux, et des analyses de propriétés sont réalisables depuis longtemps sur l'abstraction de ces objets qui ne conserve que la structure de contrôle, c'est-à-dire l'automate. Plus récemment, des techniques d'analyse approchées ont été proposées pour tenir compte des variables contrôlées par l'automate [Hal93, HMP95, Hal]. L'automate interprété est d'autre part utilisé pour la génération de code C, par exemple. Ce schéma justifie l'expression WYPIWYE (What You Prove Is What You Execute) de Gérard Berry, puisque le même modèle est utilisé comme support de la vérification et comme étape intermédiaire de la génération de code. Il suffit de se convaincre que l'algorithme de codage de l'automate est suffisamment simple pour qu'on soit sûr de son implantation.

L'hypothèse de synchronisme est la base sémantique commune de langages aux styles très différents (impératif pour ESTEREL, avec des structures de contrôle temporelles comme le chien de garde, déclaratif flot de données pour LUSTRE, ...). L'essentiel de ce document porte sur l'étude et l'implantation d'un langage de la famille synchrone, dont le style est inspiré des constructions d'automates des Statecharts.

1.4 Le langage ARGOS et son environnement de programmation

1.4.1 Description

ARGOS est un langage synchrone à syntaxe potentiellement graphique, qui permet la description de systèmes réactifs sous forme de compositions de machines de Mealy. Le jeu d'opé-

rateurs est en quelque sorte minimal : mise en parallèle, synchronisation par le mécanisme de *diffusion synchrone* commun à tous les langages synchrones, composition hiérarchique héritée des Statecharts. La composition parallèle de machines à états — que l'on trouve déjà dans SDL [CCI89] par exemple — et la composition hiérarchique, rendent réaliste la description de systèmes de taille conséquente par des systèmes d'états et de transitions explicites.

Dès le départ, le langage et l'environnement de programmation correspondant Argonaute ont été définis avec le parti pris de permettre la connexion au maximum d'outils traitant des systèmes de transitions, en particulier les outils de vérification développés à Vérimag ou dans d'autres équipes avec lesquelles nous entretenons des collaborations étroites. L'environnement Argonaute est actuellement connecté, entre autres, à : Aldébaran (calcul de relations d'équivalence sur des systèmes de transitions étiquetées, Vérimag) ; Kronos (évaluation de formules de logique temporelle temps-réel, Vérimag) ; Mec (évaluation de formules du μ-calcul, LaBri, Bordeaux) ; Autograph (placeur interactif de systèmes de transitions, INRIA Sophia-Antipolis) ; Bac (analyse de systèmes d'équations booléennes, Vérimag) ; Polka (analyse approchée des automates hybrides, Vérimag) ; Toupie (résolution de contraintes sur des entiers bornés — entre autres —, LaBri, Bordeaux).

Beaucoup plus récent que les autres langages synchrones, qui sont utilisés dans l'industrie, Argos ne dispose pas d'un environnement de programmation directement transférable. Cela est dû, en grande partie, à la nécessité de fournir un éditeur syntaxique graphique, pour l'utilisation en vraie grandeur et par des non spécialistes. En revanche sa simplicité de structure et la facilité de connexion de l'environnement Argonaute aux prototypes universitaires implantant les méthodes de vérification les plus récentes, en font un outil de travail précieux, et un bon moyen de démontrer les apports de l'approche synchrone dans des domaines où elle est encore peu utilisée.

1.4.2 Historique des travaux

Première étude des Statecharts

Les travaux autour d'ARGOS ont débuté comme une tentative de définir une algèbre de processus inspirée des Statecharts, compatible avec la technologie de vérification développée dans le projet Spectre, et intégrable dans les environnements de programmation et validation alors en cours de réalisation. Cette première étude fait l'objet de mon mémoire de DEA, soutenu en 1987. L'objet d'étude ne s'appelle pas encore ARGOS, ce n'est alors pas encore un langage synchrone, ni même à proprement parler un langage. A l'époque, la communauté de recherche internationale n'a pas encore donné aux Statecharts les 22 sémantiques qu'ont leur connaît maintenant [vdB94]. La conséquence première de cette année de travail est la constatation d'une impossibilité : les Statecharts, tels qu'ils sont décrits alors dans un rapport publié ensuite dans [Har87], forment un ensemble trop riche pour être formalisé simplement ; d'autre part, certaines des constructions offertes rendent le comportement des programmes ambigu ou imprévisible. La possibilité de structuration hiérarchique des descriptions à base d'automates semble pourtant mériter encore un peu d'attention.

Définition d'un langage de la famille synchrone

La première définition du langage synchrone ARGOS, et le premier environnement de programmation associé ARGONAUTE, remontent à ma thèse, en janvier 1990. L'éditeur graphique est dû à quatre étudiants du DESS Génie Informatique de Grenoble. Il est réalisé en X10, qui devient

rapidement obsolète. A partir de 1988, et grâce à l'influence conjuguée de G. Berry et N. Halbwachs, la poursuite du travail autour d'un langage à base d'automates intégrant la structure hiérarchique des Statecharts se rapproche inexorablement du monde des langages synchrones. La définition d'ARGOS publiée dans ma thèse montre toutefois quelques réminiscences des algèbres de processus, dont le mode de communication par rendez-vous et l'indéterminisme intrinsèque des descriptions, qui se marient mal avec les aspects synchrones de la composition parallèle.

Le travail se poursuit ensuite en collaboration étroite avec Muriel Jourdan, qui effectue sous ma direction un stage de magistère, un DEA et une thèse, couvrant ainsi la période juin 1990 — septembre 1994.

La programmation multi-langages

Dès le début de la thèse de Muriel Jourdan en 1991, les travaux "de fond" autour d'ARGOS et des sémantiques synchrones — définition de nouvelles constructions, généralisation du langage, étude d'exemples, implantation, élaboration d'un cadre sémantique commun permettant de comparer différents modes de synchronisation — sont conduits en parallèle avec une nouvelle direction de recherche : la programmation synchrone "multi-styles" ou multi-langages.

L'idée provient d'une constatation simple : les divers styles de programmation offerts par les langages synchrones existants — impératif textuel, flot de données, automates — correspondent à des cultures différentes dans l'industrie. D'autre part les styles *flot de données* et *automates*, par exemple, se révèlent également insuffisants lorsqu'ils sont utilisés seuls. Deux possibilités sont alors envisageables : définir un nouveau langage inspiré des langages existants, en associant des constructions de styles différents — c'est la programmation "multi-styles" ; ou profiter de l'existence des langages et de leurs environnements de programmation associés, en particulier les compilateurs, pour construire un environnement dans lequel des portions de programme de différents langages peuvent cohabiter — c'est la programmation "multi-langages".

Le souci de ne pas ré-inventer la roue et la disponibilité réduite d'un enseignant pour les développements logiciels conséquents nous ont dès le début conduits à choisir la deuxième possibilité, avec une contrainte forte : proposer l'intégration au niveau source des différents langages, tout en utilisant les compilateurs d'origine. Les chercheurs de l'équipe d'Axel Poigné à la GMD (...) semblent suivre depuis 2 ans une voie intermédiaire : ils proposent d'intégrer ARGOS, LUSTRE et ESTEREL dans un même environnement de programmation, et, tout en respectant scrupuleusement leur définition, développent un nouveau compilateur pour chacun de ces langages. Outre la duplication de travail, cette démarche pose le problème général des dialectes de langages, dont la communauté synchrone semblait jusqu'ici avoir été préservée.

La démarche que nous suivons, si elle évite de lourds investissements en développement logiciels, pose des problèmes théoriques intéressants. Après avoir travaillé essentiellement sur le mélange LUSTRE+ARGOS, nous nous sommes penchés récemment sur ARGOS+ESTEREL, sur la suggestion de G. Berry. Charles André, de l'Université de Nice, travaille pour sa part à la définition de SyncCharts [And96, AG95], un langage graphique inspiré du Grafcet et des Statecharts, et compilé en ESTEREL. Les problèmes rencontrés et les solutions envisagées sont très similaires.

Extensions temporelle et hybride

Quelque temps plus tard, vers la fin de 1992, un nouveau thème apparaît. Une partie de l'équipe commence à travailler sur le modèle des automates temporisés (*Timed Graphs* [AD90])

pour la modélisation des systèmes temporisés. Xavier Nicollin définit l'algèbre ATP (*Algebra of Timed Processes*) et sa sémantique en termes d'automates temporisés. Nous proposons alors une extension temporelle d'ARGOS et sa sémantique en termes d'automates temporisés. L'extension temporelle s'inspire d'une construction de temporisation des états proposée dans l'un des premiers rapports de recherche sur les Statecharts datant de 1984, mystérieusement disparue ensuite.

Ce travail est ensuite intégré à la thèse de Muriel Jourdan, intitulée : *Un environnement multi-langages et multi-outils pour la programmation et la vérification des systèmes réactifs.* M. Jourdan intègre l'extension temporelle au compilateur ARGOS existant, et nous commençons à expérimenter la modélisation de systèmes en ARGOS temporisé. Le compilateur produit des automates temporisés au format d'entrée de l'outil KRONOS développé dans l'équipe par S. Yovine et A. Olivero. KRONOS permet d'évaluer des formules de logique temporelle temps-réel sur un automate temporisé et implante un algorithme d'évaluation symbolique défini par X. Nicollin et J. Sifakis. L'utilisation du compilateur ARGOS couplé avec KRONOS trouve une application intéressante dans le stage post-doctoral que Muriel Jourdan effectue dans l'équipe de Bernard Espiau en 1994-95. Il s'agit de modéliser des schémas de synchronisation de tâches-robots en ARGOS temporisé, afin de détecter des blocages. Le travail s'intègre dans le projet ORCCAD [SECK93] (Open Robot Controller Computer Aided Design) de l'INRIA.

Pour cette première extension d'ARGOS, nous avons redéfini la sémantique des constructions, qui expriment maintenant des compositions de "machines de Mealy temporisées". Ce sont des automates dont les transitions et les états peuvent porter des informations temporelles. Cette nouvelle sémantique et sa propriété de compositionnalité s'inspirent bien sûr des travaux antérieurs, et ne représentent donc pas un travail théorique démesuré, mais il apparaît finalement peu satisfaisant d'avoir à ré-écrire ainsi l'essentiel de la sémantique des opérations, tout en s'attachant à résoudre les nouveaux problèmes que pose l'extension temporelle.

Généralisation : ARGOS comme constructeur d'automates décorés

Ce souci conduit alors assez naturellement à l'idée suivante : peut-on considérer ARGOS comme un jeu d'opérateurs permettant de construire de gros automates à partir d'automates plus petits, et ceci indépendamment de la nature des informations portées par les états et les transitions ? Nous appliquons cette idée à la modélisation des systèmes dits *hybrides*, dans lesquels des composants discrets contrôlent une activité continue. Dans le cas général l'activité continue est décrite par des systèmes d'équations différentielles, qui constituent un modèle hors de portée des capacités d'analyse de propriétés par des méthodes informatiques. Nous utilisons un modèle des systèmes hybrides moins expressif, qui associe aux états et transitions d'un automate des systèmes de contraintes linéaires. Ce modèle est analysable par des méthodes approchées développées par N. Halbwachs.

Ce travail aboutit à la réalisation du prototype HARGOS, qui permet de décrire des programmes ARGOS en associant aux états et transitions des automates de base des "*pragmas*" de nature quelconque. Ces pragmas ne sont pas analysés par le compilateur, qui se contente de produire un unique automate (le résultat de la composition) dont les états et les transitions portent des compositions de pragmas. N. Halbwachs développe la connexion d'HARGOS à son outil POLKA d'analyse de systèmes de contraintes linéaires. Un sous-produit de ce travail est un nouveau compilateur d'ARGOS temporisé en automates temporisés, qui rentre maintenant dans un cadre plus général.

Non déterminisme et langages synchrones

C'est en traitant des exemples de systèmes hybrides grâce au prototype HARGOS que nous commençons à nous poser des questions de fond sur la signification du non déterminisme dans les langages synchrones, et sur le bien fondé de la technique des oracles pour simuler le non déterminisme des systèmes hybrides en ARGOS. Le chapitre 4 présente les problèmes de représentation du non déterminisme, en particulier pour des exemples issus de la modélisation des systèmes hybrides. Il s'agit d'une étude centrée sur ARGOS, mais qui clarifie les aspects de non déterminisme dans les langages synchrones, et est applicable à d'autres langages.

Sémantique en systèmes d'équations

La première étude du langage ARGOS+LUSTRE avait déjà conduit à définir une traduction structurelle d'ARGOS en LUSTRE, c'est-à-dire en équations. Nous avons ensuite retravaillé cette traduction pour obtenir un programme au format d'entrée de l'outil Bac (*Boolean Automaton Checker*) développé par N. Halbwachs pour étudier les problèmes de causalité [HM95].

Finalement, le format commun DC [CS95] a été défini. Il s'agit d'un format équationnel commun à tous les langages synchrones, pour lequel des outils communs seront définis. L'environnement de programmation actuel d'ARGOS est basé sur une première phase de traduction d'ARGOS en systèmes d'équations, modèle sur lequel de nombreuses manipulations peuvent être effectuées de manière symbolique. La traduction en DC est ensuite facile, de même que la connexion à Bac.

1.4.3 Travaux en cours et perspectives

Les différentes directions de recherche suivies, qui auraient pu conduire à la dispersion du travail, semblent maintenant converger.

Nous avons défini une syntaxe textuelle pour ARGOS, dans laquelle figurent les constructions de base, les constructions nécessaires à l'extension temporisée, la possibilité d'attacher des pragmas de nature quelconque aux états et transitions des automates de base, et diverses autres extensions. La sémantique repose toutefois sur le jeu d'opérateurs primitif décrit au chapitre 2. La démarche qui consistait à simplifier le plus possible le formalisme des Statecharts afin de n'en garder que la hiérarchie des automates a conduit à un jeu d'opérateurs assez réduit, dont la sémantique s'exprime simplement, et qui a de bonnes propriétés comme la compositionnalité. D'autre part la plupart des constructions des Statecharts qui avaient été écartées dans un premier temps, pour leur sémantique ambiguë ou leur redondance, ont pu être réintroduites sous forme de macro-notations, sans changer, donc, le coeur du langage.

Le travail suit maintenant deux voies: l'étude et la réalisation d'un moteur synchrone de composition d'automates décorés (éventuellement non déterministes); la programmation multi-langage synchrone, et plus précisément la programmation mixte ARGOS/LUSTRE, ARGOS/ESTEREL ou même ARGOS/LUSTRE/ESTEREL, par compilation en format intermédiaire DC.

J. Buck de Synopsys a par ailleurs choisi ARGOS parmi toutes les variantes de langages et de sémantiques proposées pour les Statecharts. ARGOS est utilisé comme base sémantique d'un langage de description de circuits offrant des compositions hiérarchiques d'automates, et traduit en ESTEREL. Les extensions proposées suivent les idées exposées au chapitre 9.

L'équipe d'Axel Poigné à la GMD développe un environnement de programmation multi-langages en utilisant LUSTRE, ARGOS et ESTEREL.

1.5 Organisation du document

Le document ne suit pas un ordre chronologique. Après cette brève introduction générale au domaine de la modélisation et de la validation des systèmes réactifs, et à l'approche synchrone, il est organisé autour de quatre thèmes :

- La définition d'ARGOS : Je présente la définition actuelle, figée depuis environ 4 ans, et sur laquelle s'appuient les autres parties. Je ne reviendrai pas sur les différentes étapes de définition du langage. D'autre part une certaine déformation professionnelle d'enseignant[3], jointe à quelques soucis pédagogiques apparus lors de la présentation du langage à des publics peu familiers des arcanes de la sémantique opérationnelle à la Plotkin, ont conduit à remettre cent fois sur le métier la formulation de la sémantique d'ARGOS. Dans ce document, je présente les différents thèmes de manière unifiée, en utilisant la formulation à mon avis la plus satisfaisante, c'est-à-dire la plus récente. Ceci explique la différence de forme entre les articles joints et ce document, mais devrait permettre une meilleure compréhension globale.

  L'étude d'ARGOS a conduit à deux travaux théoriques, motivés rapidement et présentés dans cette partie : une solution à l'introduction du non déterminisme dans un langage synchrone ; un cadre général pour comparer des modes de communication synchrone dans les algèbres de processus ou les langages.

  Je présente également dans cette partie quelques réflexions relatives à la syntaxe concrète du langage.

- La programmation multi-langages : on trouve dans cette partie un rappel des premiers travaux sur le sujet, qui ont conduit aux publications [JLMR94, JLMR93] et sont exposés en détail dans la thèse de Muriel Jourdan [Jou94]. J'expose ensuite l'état actuel des travaux sur la programmation ARGOS+LUSTRE via DC, qui offre l'avantage de reporter sur le format DC le problème de détection des programmes non causaux, où il semble être plus facile à résoudre. On lira également un chapitre sur les propositions d'intégration ARGOS+ESTEREL, lesquelles ressemblent énormément aux propositions de Charles André pour les *SyncCharts*.

- Les extensions temporelle et hybride : cette partie reprend rapidement les motivations et les résultats de [JMO93], et présente plus en détail les applications du couplage ARGOS temporisé + KRONOS, dans une application en robotique.

- L'environnement de programmation ARGONAUTE : J'aborde ici les aspects d'édition syntaxique graphique du langage ; de compilation vers diverses représentations des systèmes de transitions ; de connexion à divers outils de vérification.

La dernière partie présente quelques unes de mes activités d'enseignement et s'attache à illustrer les influences réciproques entre enseignement et recherche. Le chapitre 14 intitulé *Automates et programmation par événements en DEUG* reprend la discussion sur la mémoire nécessaire au calcul des "sorties", pour l'étude des algorithmes itératifs.

3. lequel doit chaque année se prémunir contre l'ennui de présenter deux fois le même sujet de la même manière

1.6 Sources

La plupart des chapitres correspondent à des articles publiés, à des travaux rédigés sous forme de rapports de recherche, à des documentations techniques, ou à des mémoires d'étudiants. L'intention des paragraphes d'introduction est de replacer l'article dans le contexte général, de reconnaître les apports extérieurs, et de compléter le contenu technique par des commentaires souvent tirés des réactions aux présentations orales. La première partie est détaillée, de manière à rendre la suite compréhensible. Les 3 autres parties sont surtout construites comme un guide de lecture des articles.

Les articles les plus significatifs sont reproduits en annexe.

Le chapitre 14 est adapté de l'ouvrage *Cours d'informatique : langages et programmation*[4], qui constitue le support d'une grande partie des enseignements d'informatique du DEUG mention MIAS (Mathématiques, Informatique et Applications aux Sciences) de Grenoble.

Pour la partie recherche, le texte de ce document est tiré des articles en anglais. Le texte est bien évidemment traduit — et souvent détaillé pour replacer les différents points techniques dans un contexte plus général. Toutefois je n'ai pas traduit les parties mathématiques, ni le texte des figures. Ainsi l'ensemble des états des machines de Mealy s'appelle-t-il S, pour *states*. Ceci procède d'un choix délibéré : j'ai préféré éviter d'introduire des erreurs ou incohérences dans les parties formelles présentées ici, par des renommages mal contrôlés. Je prie toutefois le lecteur de m'en excuser.

4. **Cours d'informatique : langages et programmation**. P.-C. Scholl, M.-C. Fauvet, F. Lagnier et F. Maraninchi, Masson 1993

Première partie

Définition d'Argos

Chapitre 2

Les constructions d'Argos

Sources : *La première sémantique qui correspond à la version du langage décrite ici fait l'objet de [Mar92]. Elle est reprise dans [Jou94]. La formulation qui fait apparaître un ensemble d'opérations sur des machines de Mealy est présentée dans [MH96a].*

Comme nous l'avons vu en introduction, un noyau réactif peut être décrit par une machine d'états finis, où les états représentent les valeurs possibles de la mémoire nécessaire au calcul des sorties. En ARGOS, les composants élémentaires des programmes sont de telles machines, dont on donne explicitement l'ensemble des états et l'ensemble des transitions. Le coeur du langage est un ensemble d'opérations qui permettent de composer des machines de Mealy booléennes. On y trouve bien sûr le produit synchrone, qui permet de représenter la mise en parallèle ; une opération d'encapsulation qui permet de définir la portée des signaux d'entrée/sortie échangés entre les différentes machines ; une opération de composition hiérarchique, ou raffinement, directement inspirée de la hiérarchie des états dans les Statecharts. Les composants qui évoluent en parallèle, ou participent à une composition hiérarchique, se synchronisent par *diffusion synchrone* de signaux.

Notons tout de suite que cette vision des choses marque une différence fondamentale avec les Statecharts, où la possibilité de tracer des transitions "inter-niveaux" empêche l'identification de composants indépendants dans un programme. Les sémantiques sont en général décrites globalement. Il n'y a pas de syntaxe qui permettrait de décrire l'ensemble des programmes de manière inductive, et donc pas de sémantique basée sur la syntaxe, dans laquelle la sémantique d'un objet composé dépend de la sémantique des composants.

Nous présentons tout d'abord les machines de Mealy booléennes et les opérations utilisées. Puis nous donnons la syntaxe d'un langage d'expressions faisant intervenir ces opérations. La sémantique de ce langage fait apparaître une notion de programme incorrect, certaines combinaisons d'opérations étant dépourvues de sens. Nous terminons par l'énoncé de la propriété de compositionnalité de cette sémantique.

La construction d'un véritable *langage*, à partir de cette base théorique, est une étape supplémentaire, qui fait l'objet du chapitre 6.

2.1 Exemple commenté

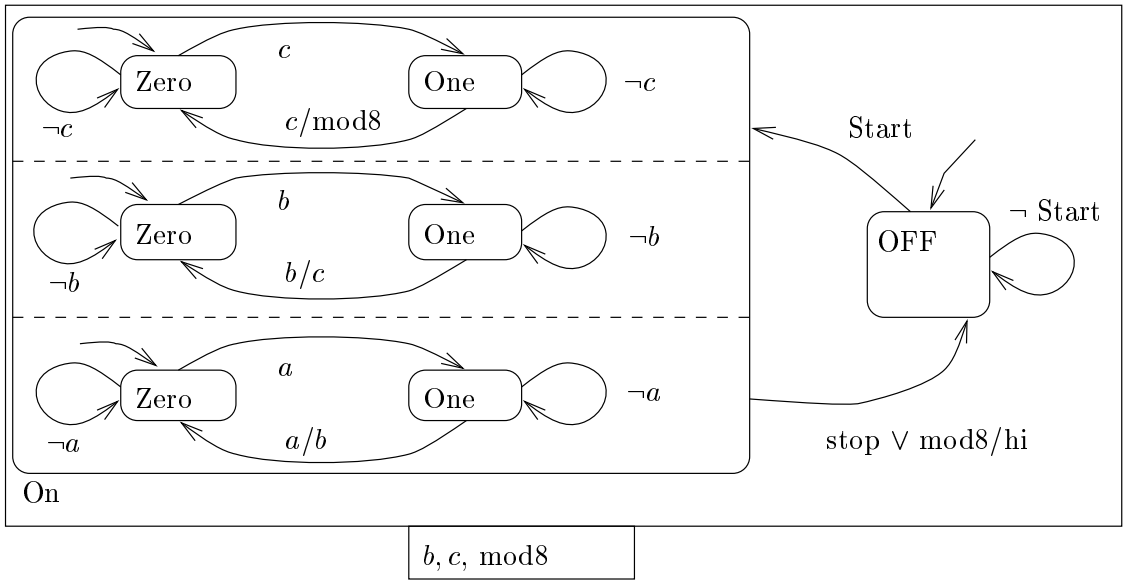


FIG. 2.1 – *Un compteur de 'a' modulo 8*

La figure 2.1 donne un programme ARGOS de comptage des signaux d'entrée 'a', avec possibilités d'arrêt et de ré-initialisation. Puisque le système est à mémoire bornée, il s'agit nécessairement d'un compteur borné. On a représenté ici un compteur modulo 8.

La composante de comptage proprement dite est réalisée par la coopération de trois compteurs 1 bit synchronisés sur les retenues. Le compteur démarre à la reception du signal Start, et s'arrête à la réception de Stop, ou lorsqu'il a accompli un tour complet (émission du signal interne mod8). Dans ce dernier cas, il émet le signal hi vers l'extérieur.

Les opérations utilisées sont : la *mise en parallèle* des trois compteurs 1 bit ; le *raffinement* de l'état de comptage actif par le compteur proprement dit ; la déclaration des signaux `b, c, mod8` comme locaux à l'ensemble du programme décrit.

Tout programme ARGOS est construit à l'aide de ces trois opérations.

2.2 Machines de Mealy booléennes et opérations

Les points d'interaction élémentaires entre le système et son environnement sont appelés *signaux*. Nous décrivons ici des systèmes capables de traiter des signaux booléens. (On peut imaginer des signaux valués par des réels, par exemple, qui représentent la lecture de capteurs physiques. ESTEREL manipule explicitement des signaux valués de types quelconques ; en LUSTRE également, les types des flots d'entrée-sortie sont quelconques). Nous considérons un ensemble de noms de signaux $\mathcal{A}$.

2.2.1 Machine de Mealy booléenne

Definition 2.1 : Machine de Mealy booléenne
Une machine de Mealy est un n-uplet (S, s_0, I, O, T) où $I \subseteq \mathcal{A}, O \subseteq \mathcal{A}$ sont les ensembles de signaux d'entrée et de sorties ; S est l'ensemble des états ; s_0 est l'état initial ; $T \subseteq S \times \mathcal{B}(I) \times 2^O \times S$ est l'ensemble des transitions. L'ensemble des machines de Mealy booléennes ainsi définies est noté M. □

$\mathcal{B}(I)$ dénote l'ensemble des formules booléennes à variables dans I. Sans perte de généralité, on peut considérer que les formules booléennes qui décrivent les conditions d'activation des transitions sont réduites à des monômes.

La figure 2.2 donne un exemple de système simple décrit par un automate. Les noms d'états n'ont pas de sémantique ; l'état initial est distingué par une flèche sans état source ; la structuration des étiquettes de transitions en entrée/sortie est marquée par un caractère '/'. Pour des détails plus complets sur la syntaxe concrète graphique utilisée, voir section 6.

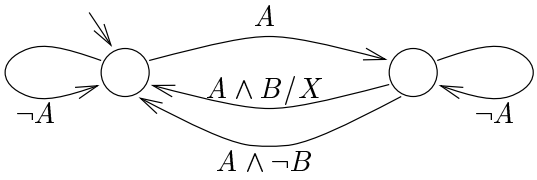


FIG. 2.2 – *Un système simple décrit par une machine de Mealy booléenne : le système émet X tous les deux A, à condition que B soit présent au même instant.*

Pour ces objets, définis syntaxiquement, on fournit une relation d'équivalence qui précise quels objets représentent en réalité le même comportement réactif, bien qu'ayant des définitions syntaxiquement distinctes. Ici c'est la *bisimulation* (pour des objets déterministes, cette relation coïncide d'ailleurs avec l'équivalence de trace).

Definition 2.2 : Bisimulation de machines de Mealy booléennes
Deux machines $M_1 = (S_1, s_{01}, I, O, T_1)$ et $M_2 = (S_2, s_{02}, I, O, T_2)$ sont dites *bisimilaires*, et on note $M_1 \approx M_2$, si et seulement si il existe une relation d'équivalence $\mathcal{R} \subseteq S_1 \times S_2$ telle que $s_{01} \mathcal{R} s_{02}$ et

$$s \mathcal{R} s' \implies \begin{cases} s \xrightarrow{b/o} r \implies \begin{cases} \exists b_1, \ldots, b_m, r'_1, \ldots, r'_m \text{ telle que} \\ \quad \forall i \in [1,m],\ s' \xrightarrow{b_i/o} r'_i \ \wedge\ r \mathcal{R} r'_i \\ \quad \wedge\ b \Rightarrow (\bigvee_i b_i) \end{cases} \\ \textit{et inversement.} \end{cases}$$

□

Cette notion de bisimulation est une extension de l'équivalence de processus booléens présentée dans [CGS91], dans laquelle nous prenons en compte des étiquettes structurées en entrées/sorties. Dans cette équivalence, on peut faire correspondre à une transition d'un système un ensemble de transitions de l'autre système. La relation est étendue aux états :

Definition 2.3 : Equivalence d'états
Soient s, s' deux états d'une machine de Mealy booléenne (S, s_0, I, O, T). Par définition $s \approx s'$ ssi $(S, s, I, O, T) \approx (S, s', I, O, T)$. □

Les machines utilisées comme composants de programmes ARGOS doivent satisfaire aux deux propriétés de *déterminisme* et *réactivité* définies par :

Definition 2.4 : Déterminisme et réactivité
Une machine (S, s_0, I, O, T) est dite *réactive* si et seulement si :

$$\forall s \in S, \left[\bigvee_{(s,m,o,s') \in T)} m \right] = \texttt{true}$$

Elle est dite *déterministe* si et seulement si :

$$\forall s \in S, \forall t_1 = (s, m_1/o_1, s_1), \forall t_2 = (s, m_2/o_2, s_2), m_1 = m_2 \quad \Longrightarrow \quad (o_1 = o_2) \wedge (s_1 \approx s_2)$$

□

On notera respectivement $\mathcal{M}^r$, $\mathcal{M}^d$ et $\mathcal{M}^{rd}$ les ensembles de machines *réactives*, *déterministes* et *réactives et déterministes*.

On peut rapprocher la notion de réactivité de ce que les concepteurs des IO-automates [LT89] appellent *"input-enabling"*. En d'autres termes, une machine réactive est capable de réagir à toute configuration des entrées, dans chacun de ses états. La réactivité (définie par rapport aux entrées) est fondamentale dans un modèle de description des systèmes réactifs, puisqu'elle symbolise le fait que l'environnement est libre. En revanche le déterminisme n'est requis que si l'on a pour but de *programmer* des systèmes réactifs. Le chapitre 4 discute en détails l'introduction du non-déterminisme dans les langages synchrones.

2.2.2 Produit synchrone de machines de Mealy booléennes

La figure 2.3 donne un programme parallèle sans synchronisation entre les composants. Le système émet un A tous les deux a, et de même un B tous les deux b. Le produit synchrone des deux composants est donné figure 2.4. C'est une opération commutative et associative.

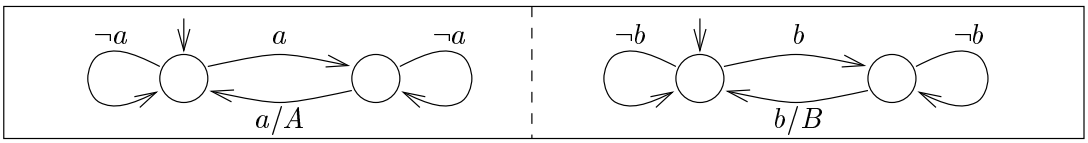


FIG. 2.3 – *Une composition parallèle sans synchronisation*

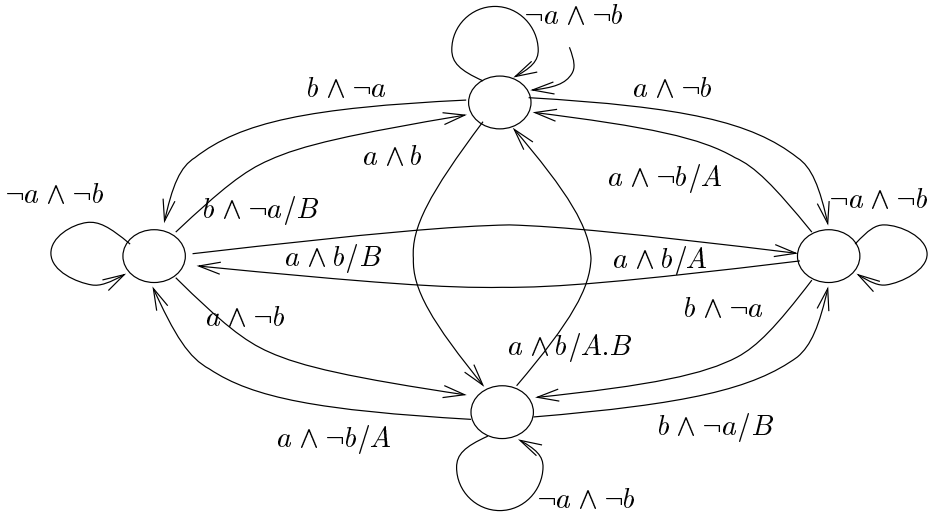


FIG. 2.4 – *Expansion du produit synchrone*

Cette opération est décrite formellement par :

Definition 2.5 : Produit synchrone

$\times \quad : \quad \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}.$

$(S_1, s_{01}, I_1, O_1, T_1) \times (S_2, s_{02}, I_2, O_2, T_2) = (S_1 \times S_2, (s_{01}, s_{02}), I_1 \cup I_2, O_1 \cup O_2, T')$

Où T' est défini par :

$((s_1, m_1, o_1, s_1') \in T_1, (s_2, m_2, o_2, s_2') \in T_2) \implies ((s_1, s_2), m_1 \wedge m_2, o_1 \cup o_2, (s_1', s_2')) \in T')$ □

Il est aisé de montrer que le produit synchrone préserve le déterminisme et la réactivité des composants.

2.2.3 Encapsulation

L'encapsulation n'a de sens que pour forcer la synchronisation entre deux composants. Deux composants X et Y peuvent être synchronisés s'ils fonctionnent en parallèle et si une sortie de l'un est une entrée de l'autre. Pour déclarer qu'un signal ne doit être utilisé qu'aux fins de synchronisation entre deux composants, on peut le déclarer *local* à l'ensemble de ces deux composants. Cette opération a un effet de renommage usuel, comme les déclarations de variables locales à un bloc dans un langage séquentiel classique (deux occurrences du même nom de signal, sous la portée de deux opérations de déclaration distinctes, représentent des signaux différents). D'autre part, un signal *local* n'est pas diffusé à l'exterieur de la zone définie par l'opération qui le déclare, et ne peut pas non plus provenir de l'extérieur de cette zone. Par conséquent, dans la portée d'un opérateur qui déclare un signal x, une transition dont la condition nécessite la présence de x ne peut être exécutée que si x est produit dans la même zone, durant la même réaction. (Cette opération décrit une membrane étanche dans les deux sens ; elle est analogue à la construction `signal ... in ... end` d'ESTEREL, et à la déclaration de flots locaux à un noeud en LUSTRE. La sémantique des Statecharts de [HGdR88] propose deux opérations — fermeture et abstraction — qui représentent les deux membranes semi-perméables correspondantes).

La figure 2.5 donne un système qui émet un A tous les 4 a ; c'est un compteur 2 bits. Il est composé de deux compteurs 1 bit qui se synchronisent sur le signal ℓ.

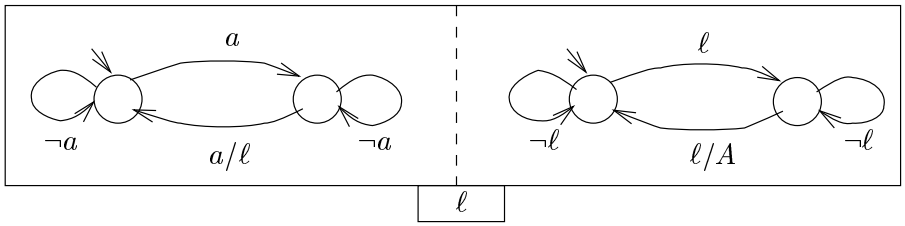


FIG. 2.5 – *Synchronisation et signaux locaux*

La figure 2.6 est une expansion partielle du programme de la figure 2.5, où seule la composition parallèle a été prise en compte. On construit toutes les combinaisons possibles d'une transition du premier processus et d'une transition du second, et l'étiquette est construite en prenant la conjonction des conditions et l'union des ensembles de signaux émis.

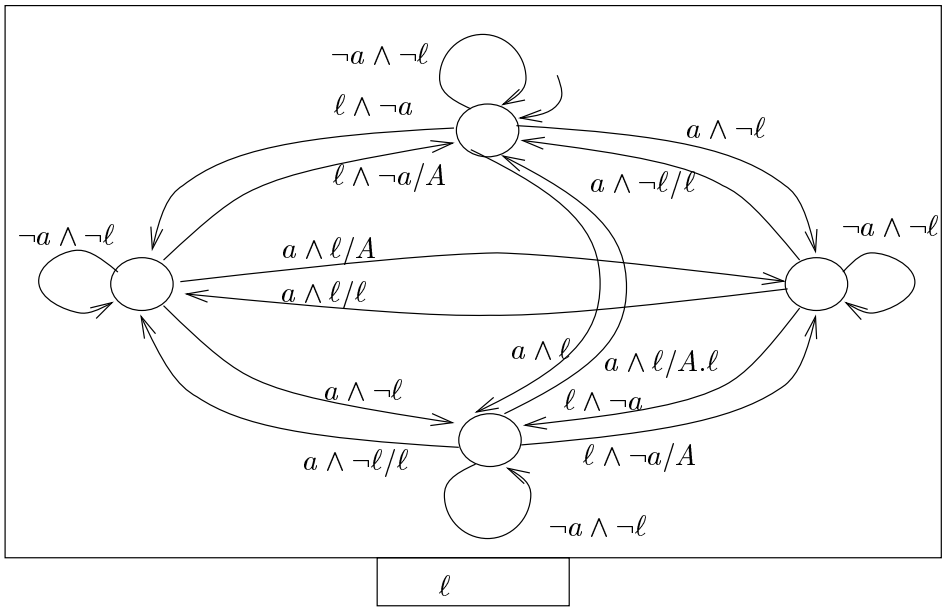


FIG. 2.6 – *Expansion de la composition parallèle*

La figure 2.7 donne la machine obtenue par application de l'opération d'encapsulation. Notez que a reste la seule entrée du système.

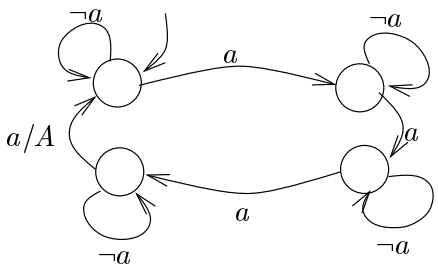


FIG. 2.7 – *Expansion de l'opération d'encapsulation*

Nous donnons la définition de l'opération d'encapsulation en considérant que la partie condition des étiquettes de transitions est réduite à un monôme, par souci de simplicité, et sans perte de généralité.

Definition 2.6 : Encapsulation
$\setminus \; : \quad \mathcal{M} \times 2^{\mathcal{A}} \longrightarrow \mathcal{M}$

$(S, s_0, I, O, T) \setminus \Gamma = (S, s_0, I \setminus \Gamma, O \setminus \Gamma, T')$, où T' est défini par :
$(s, m, o, s') \in T \;\wedge\; m^+ \cap \Gamma \subseteq o \;\wedge\; m^- \cap \Gamma \cap o = \emptyset \quad \Longrightarrow \quad (s, \exists \Gamma.m, o \setminus \Gamma, s') \in T'$ □

m^+ dénote l'ensemble des variables qui apparaissent positives dans le monôme m (c'est-à-dire $m^+ = \{x \in \mathcal{A} \mid (x \wedge m) = m\}$). m^- dénote l'ensemble des variables qui apparaissent négatives dans le monôme m (c'est-à-dire $m^- = \{x \in \mathcal{A} \mid (\neg x \wedge m) = m\}$).

Intuitivement, une transition $(s, m, o, s') \in T$ subsiste dans la machine résultat de l'encapsulation si son étiquette satisfait à un critère local : $m^+ \cap \Gamma \subseteq o$, qui signifie que tout signal local supposé présent doit être effectivement émis par la transition ; et $m^- \cap \Gamma \cap o = \emptyset$, qui signifie que tout signal local supposé absent ne doit pas être émis. D'autre part, si l'étiquette de transition satisfait ce critère, elle est modifiée pour cacher les signaux locaux.

L'opération d'encapsulation peut ne pas préserver le déterminisme ou la réactivité de son opérande.

Le chapitre 3 reprend l'essentiel de l'article [JM94b], où cette opération d'encapsulation est généralisée dans un cadre formel qui permet d'exprimer la sémantique de plusieurs calculs synchrones. Le calcul de Milner [Mil89] et les "réactions en chaîne" des Statecharts [PS88] peuvent être décrits par une opération d'encapsulation en variant le critère local et la fonction utilisée pour masquer les objets locaux.

2.2.4 Encapsulation et systèmes d'équations

Il n'est pas forcément très facile de se convaincre que la définition de l'opération d'encapsulation ci-dessus correspond à l'idée intuitive de diffusion synchrone des signaux. Une définition de la diffusion synchrone à l'aide de systèmes d'équations booléennes est plus naturelle. L'exemple de la figure 2.8 illustre la correspondance.

Calcul de l'encapsulation par résolution d'un système d'équations booléennes

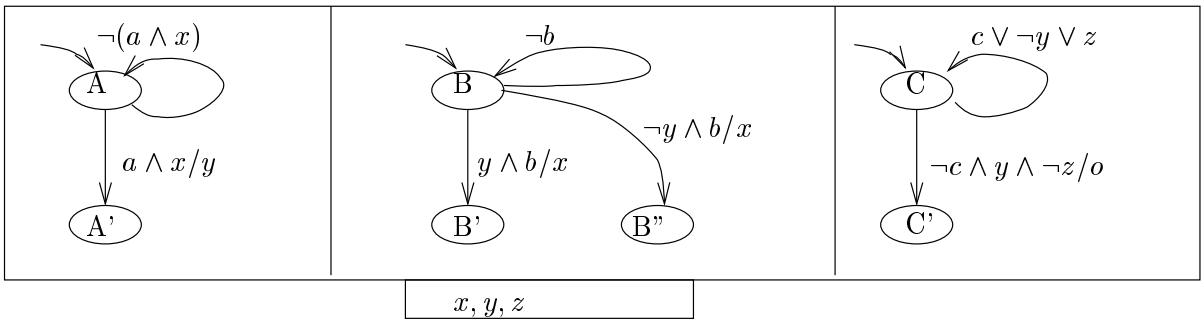


FIG. 2.8 – *Composition parallèle, encapsulation et systèmes d'équations*

Lorsque le système est dans l'état ABC, et que l'entrée $a \wedge b \wedge \neg c$ survient, la réaction globale est déterminée par le statut des signaux locaux x, y et z. Ce statut doit être unique: dans une réaction du système, un signal local est présent ou absent, mais ne change pas de statut. D'autre part, on peut écrire le système d'équations suivant:

$$\begin{aligned} x &= b \wedge y \ \vee \ b \wedge \neg y \\ y &= a \wedge x \\ z &= \text{FALSE} \\ a = b &= \text{TRUE}, c = \text{FALSE} \end{aligned}$$

pour exprimer, entre autres, que: x *est présent à condition qu'une transition qui émet* x *soit exécutée*, c'est-à-dire lorsque $b \wedge y$ ou lorsque $b \wedge \neg y$. Le système a ici une solution unique: $x = y = \text{TRUE}$, $z = \text{FALSE}$. En reportant cette information sur le programme, on déduit une transition globale de l'état ABC à l'état $A'B'C'$, étiquetée par $a \wedge b \wedge \neg c/o$.

Un tel système, pour un état et une configuration des entrées donnés, peut ne pas avoir de solutions, ou en avoir plusieurs. S'il en a plusieurs, c'est que le statut des signaux locaux n'est pas déterminé de manière unique, et en reportant sur le programme les diverses valuations des signaux locaux, on obtient plusieurs transitions issues du même état global, pour la même configuration des entrées: le système global n'est plus déterministe. De manière analogue, le système global n'est plus réactif, s'il existe un état et une configuration des entrées pour lesquels le système est sans solution.

Comparaison avec la définition des opérations $\times$ et $\backslash$

Le calcul de la machine de Mealy correspondant au programme de la figure 2.8 est réalisé en deux temps: l'expansion de la composition parallèle (opération $\times$), et la suppression de certaines des transitions, d'après le critère de l'encapsulation (opération $\backslash$).

Dans le système obtenu par expansion de la composition parallèle, et puisque cette composition préserve la réactivité, de tout état partent des transitions étiquetées par tous les monômes construits sur l'ensemble des signaux d'entrée x, y, z, a, b, c. Si l'on fixe une valuation v des signaux a, b, c, on doit encore trouver tous les monômes $v \wedge v'$, où v' est une valuation des signaux x, y, z. D'autre part, une transition constitue un parfait résumé des conséquences d'une hypothèse sur le statut des signaux locaux, puisqu'on trouve, dans la partie *émission* de son étiquette, tous les signaux émis, toutes composantes parallèles confondues.

Dans l'exemple ci-dessus, parmi les transitions issues de l'état ABC pour l'entrée $a \wedge b \wedge \neg c$, on trouve : $a \wedge b \wedge \neg c \wedge x \wedge y \wedge \neg z / x, y, o$, ce qui signifie que, si l'on suppose x et y présents et z absents, les transitions exécutées sont telles que l'on émet x et y mais pas z.

Avant encapsulation, les transitions issues d'un même état, pour la même configuration des signaux externes, présentent donc toutes les hypothèses sur le statut (toutes les valuations booléennes) des signaux locaux x, y, z. (En conséquence, si l'encapsulation porte sur n signaux, on trouve 2^n transitions par entrée externe et par état. Cela constitue d'ailleurs une raison suffisante pour que le compilateur ne soit pas basé sur une construction explicite des machines en suivant la sémantique présentée ici).

Le critère utilisé pour conserver ou rejeter une transition revient à déterminer si la valuation (l'hypothèse sur le statut) des signaux locaux est une solution du système d'équations correspondant. L'application des deux opérations (composition parallèle et encapsulation) est donc assimilable à la résolution du système d'équations, par essai de chaque valuation des inconnues.

2.2.5 Condition d'activation

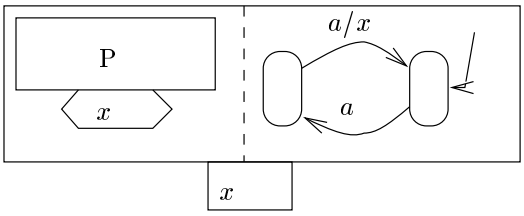
Il s'agit d'une opération unaire, paramétrée par une condition booléenne quelconque, qui exprime qu'un processus ne fonctionne que lorsque cette condition est vraie. Combinée avec un raffinement, elle permet d'exprimer la suspension de processus.

Definition 2.7 : Condition d'activation
$\triangle \quad : \quad \mathcal{M} \times \mathcal{B}(\mathcal{A}) \longrightarrow \mathcal{M}$

$(S, s_0, I, O, T) \triangle b = (S, s_0, I \cup Var(b), O, T' \cup \{(s, \neg b, \emptyset, s) | s \in S\})$
où T' est défini par: $(s, m, o, s') \in T \iff (s, m \wedge b, o, s') \in T'$

□

Exemple 2.1 : Filtrage des entrées
La définition d'une condition d'activation peut être utilisée, par exemple, pour filtrer les entrées d'un processus. Pour exprimer que le processus P ne réagit qu'à une occurrence du signal a sur deux, on peut écrire :



□

2.2.6 Raffinement, ou composition hiérarchique

Exemple 2.2 : Différents types de terminaison
Dans un raffinement, un automate joue le rôle de *contrôleur*, qui peut lancer et arrêter ses processus raffinants. Une transition issue d'un état X raffiné par P interrompt P. Toutefois cette transition peut-être provoquée par un événement issu de P. La figure 2.9 illustre les différentes

situations. *stop* et *reset* sont des interruptions du processus raffinant ; *reset* tue le processus, et en relance un exemplaire dans son état initial, au cours de la même réaction. La transition *end*, en revanche, étant déclenchée par un signal que le processus raffinant émet lui-même, peut être considérée comme une terminaison normale.

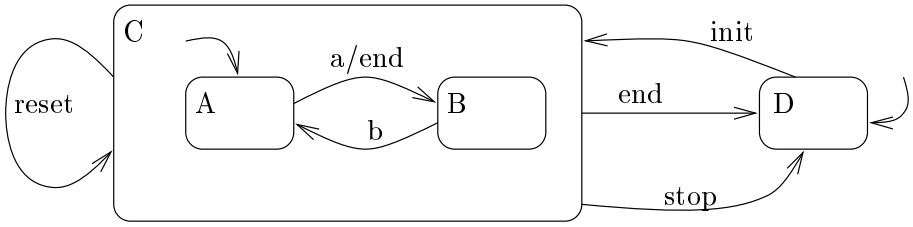


FIG. 2.9 – *Raffinement et terminaison*

□

L'opération de raffinement, dont les arguments sont un automate à n états et n machines destinées à raffiner ces états, est décrite formellement ci-dessous.

Definition 2.8 : Raffinement
$\triangleright \quad : \quad \mathcal{M} \times 2^{\mathcal{M}} \longrightarrow \mathcal{M}$
Notons $M = (S, s_0, I, O, T)$ où $S = \{s_0, s_1, ..., s_n\}$. Considérons également un ensemble $\{M_j\}_{j=0..n}$ de machines à utiliser pour raffiner les états de M, où $M_j = (S^j, s_0^j, I^j, O^j, T^j)$, et $S^j = \{s_0^j, s_1^j, ..., s_{n_j}^j\}$.
La machine composée, dans laquelle chaque M_j raffine l'état s_j correspondant, est de la forme :
$M \triangleright \{M_j\}_{j=0..n} = (S \triangleright \{S^j\}_J, s_0 \triangleright s_0^0, I \cup \bigcup I^j, O \cup \bigcup O^j, T')$.
Son ensemble d'états est de la forme :
$S \triangleright \{S^j\}_J = \bigcup_{j=0}^{n} \{s_j \triangleright s_k^j, k \in [0..n_j]\}$

Et ses transitions T' sont données par les deux règles suivantes :
1) Une transition de s_a à s_b dans la machine de contrôle, jointe à une transition de s_k^a à $s_{k'}^a$ dans la machine raffinant l'état courant. Les sorties sont rassemblées. Dans l'état but global, la machine raffinant l'état s_b est lancée dans son état initial, et la machine raffinant l'état s_a a été tuée. Son état interne n'est plus pertinent :

$$(s_a, m, o, s_b) \in T \quad \wedge \quad (s_k^a, m', o', s_{k'}^a) \in T^a \qquad \Longrightarrow \quad (s_a \triangleright s_k^a, m \wedge m', o \cup o', s_b \triangleright s_0^b) \in T'$$

2) Une transition de la machine raffinant l'état courant, de s_k^a à $s_{k'}^a$, lorsqu'aucune transition issue de s_a n'est activable dans la machine de contrôle :

$$(s_k^a, m', o', s_{k'}^a) \in T^a \quad \Longrightarrow \quad (s_a \triangleright s_k^a, m' \wedge \left[\bigwedge_{(s_a, m, -, -) \in T} \neg m \right], o', s_a \triangleright s_{k'}^a) \in T'$$

□

Le raffinement n'est pas primitif, et peut-être construit de manière structurelle à l'aide de la mise en parallèle et de la déclaration de signaux locaux. Une partie du DEA de Muriel Jourdan en

1990/91 consistait à démontrer cette propriété et à traduire ARGOS, après une phase d'élimination des raffinements, dans l'un des formats intermédiaires du compilateur ESTEREL, IC. Ce travail est implanté dans le compilateur ARGOS (voir aussi le chapitre 13).

Dans la plupart des articles où il s'agit de donner la sémantique formelle d'un jeu d'opérateurs pour prouver des propriétés de compositionnalité, ou pour étudier les problèmes de déterminisme et de réactivité, nous omettons le raffinement. Nous le présentons néanmoins comme une opération à part entière, lorsqu'il s'agit d'illustrer l'adéquation du jeu d'opérations à la description de structures courantes des systèmes, ou lorsque la présence d'opérations de raffinement explicitement voulues par le programmeur est utilisée pour améliorer une compilation d'ARGOS. C'est le cas, typiquement, lorsqu'on s'intéresse à la traduction en systèmes d'équations munies de conditions d'activation (voir chapitre 5).

Raffinement avec initialisation Le raffinement est une structure qui permet de contrôler l'activité d'un ensemble de processus. Le processus qui raffine un état X réagit toujours dans l'instant où le contrôleur le détruit en quittant l'état X. En revanche, le processus qui raffine un état X ne réagit jamais dans l'instant où le contrôleur le crée en entrant dans l'état X. Cette restriction impose des contraintes à la conception des systèmes.

Toutefois, si l'on introduit le raffinement avec initialisation du processus crée, il faut traiter les problèmes dits de "*réincarnation*", tels qu'ils apparaissent en ESTEREL dans les boucles. En effet, une transition qui boucle sur un état raffiné contrôle en fait deux instances du processus raffinant : l'une est en train de disparaître, et exécute une dernière transition ; l'autre est en cours de création, et exécute une première transition. La compilation doit tenir compte de cette possibilité d'existence de plusieurs instances d'un même objet — dont l'intersection des périodes d'existence ne couvre toutefois qu'un instant, ce qui permet de traiter le problème en évitant la duplication de code. Le problème est similaire à celui traité par P. Caspi pour l'extension récursive de LUSTRE.

Nous reviendrons sur le raffinement avec initialiation au chapitre 9 qui expose les possiblités de mélange ARGOS+ESTEREL.

2.3 Un langage d'expressions et sa sémantique

A partir de ce jeux d'opérateurs, on définit le langage $\mathcal{E}$ des programmes par une grammaire d'expressions d'axiome E, qui détermine les imbrications possibles d'opérateurs :

E	$::=$	$E \| E$	Composition parallèle
	$\mid$	$\overline{E^{\Gamma}}$	Encapsulation $\Gamma \subseteq \mathcal{A}$
	$\mid$	$\mathbf{R}_M(R_1, ..., R_n)$	Raffinement de M par les R_i
	$\mid$	E when b	Condition d'activation
R	$::=$	$E \mid$ NIL	Processus raffinants

La notation NIL est introduite pour identifier les états terminaux, non raffinés, des machines du programme.

La fonction sémantique $\mathcal{S} : \mathcal{E} \longrightarrow \mathcal{M}^{rd} \cup \{\bot\}$ est définie récursivement par :

$$\mathcal{S}(E_1 \| E_2) = \begin{cases} \bot \text{ si } \mathcal{S}(E_1) = \bot \text{ ou } \mathcal{S}(E_2) = \bot \\ \mathcal{S}(E_1) \times \mathcal{S}(E_2) \text{ sinon} \end{cases}$$

$$\mathcal{S}(\mathbf{R}_{M^d}(R_1, ..., R_n)) = \begin{cases} \bot \text{ si } \exists i \in [1, n] \text{ t.q. } \mathcal{S}(R_i) = \bot \\ M^d \triangleright (\mathcal{S}(R_1), ...\mathcal{S}(R_n)) \text{ sinon} \end{cases}$$

$$\mathcal{S}(\overline{E^{\Gamma}}) = \begin{cases} \bot \text{ si } \mathcal{S}(E) = \bot \\ \text{sinon: soit } X = \mathcal{S}(E) \setminus \Gamma \text{ dans} \quad \text{si } X \in \mathcal{M}^{rd} \text{ alors } X \text{ else } \bot \end{cases}$$

$$\mathcal{S}(E \text{ when } b) = \begin{cases} \bot \text{ si } \mathcal{S}(E) = \bot \\ \mathcal{S}(E) \triangle b \text{ sinon} \end{cases}$$

$$\mathcal{S}(\mathsf{NIL}) = (\{\mathsf{NIL}\}, \mathsf{NIL}, \emptyset, \emptyset, \{(\mathsf{NIL}, \mathsf{true}, \emptyset, \mathsf{NIL})\})$$

Un programme P est dit *incorrect* si et seulement si $\mathcal{S}(P) = \bot$. Les erreurs sont détectées lors des encapsulations, si le résultat de l'opération n'est pas réactif et déterministe. La valeur d'erreur $\bot$ se propage.

Cette notion de correction est en quelque sorte minimale. Considérant que les objets construits doivent être à la fois déterministes et réactifs, et que l'hypothèse de synchronisme — qui est à la base de l'opération d'encapsulation — peut faire apparaître des objets non déterministes ou non réactifs, on ne peut pas accepter toutes les constructions.

Cette notion de correction ne correspond pas à la notion dite "*de causalité*" introduite pour ESTEREL. Les réflexions sur une notion de *causalité* pour ARGOS étant entièrement inspirées de l'exemple d'ESTEREL, elles sont reportées au chapitre 9 "*Argos + Esterel*".

2.4 A propos d'expressivité

Toute proposition d'un jeu d'opérateurs, dans quelque domaine que ce soit, pour décrire une certaine classe d'objets, suscite une interrogation légitime : *est-ce suffisant?* Par exemple on sait qu'en supprimant la récursivité et la boucle de type "while" dans un langage structuré (sans branchements intempestifs), on réduit l'ensemble des programmes possibles. Pour formaliser cette question, on peut considérer l'ensemble des objets à décrire Y, l'ensemble E de toutes les expressions engendrées par le jeu d'opérateurs, et la fonction f qui décrit la correspondance entre expressions et objets à décrire. On a en général $f(E) \subseteq Y$. Reste à savoir si $Y \subseteq f(E)$, c'est-à-dire si f est surjective.

Dans le cas d'ARGOS exposé ci-dessus, où il s'agit de choisir un jeu d'opérateurs pour décrire des machines de Mealy booléennes, la réponse est triviale : les objets de Y, c'est-à-dire les machines de Mealy, sont des constantes du langage proposé, et, pour une machine M, $f(M) = M$. (En LUSTRE, la propriété est un peu moins triviale, on la prouve en exhibant un codage systématique des machines de Mealy en programmes LUSTRE, c'est-à-dire une fonction g de Y vers E, telle que $f(g(y)) = y$).

Nous avons signalé plus tôt que le raffinement n'est pas une opération primitive ; on voit ici que la composition parallèle ne l'est pas plus. Cette notion d'expressivité n'est donc pas très intéressante. On aimerait toutefois trouver des raisons à l'introduction de certains opérateurs comme la condition d'activation.

2.4.1 Comparaison avec ESTEREL

Une des manières de juger l'adéquation du jeu d'opérateurs ARGOS à la description des systèmes réactifs est de le comparer aux constructions d'ESTEREL, langage pour lequel un travail considérable de choix des structures adéquates a été réalisé ([Ber93] décrit un jeu d'opérateurs à l'aide duquel on peut obtenir toutes les constructions d'ESTEREL). Le chapitre 9, en décrivant les motivations d'une utilisation mixte ARGOS/ESTEREL, fournit des éléments de comparaison précis. Je donne ici un simple exemple de construction ARGOS pour exprimer l'une des structures de contrôle temporelles d'ESTEREL : l'interruption préemptive (le "watching" fort).

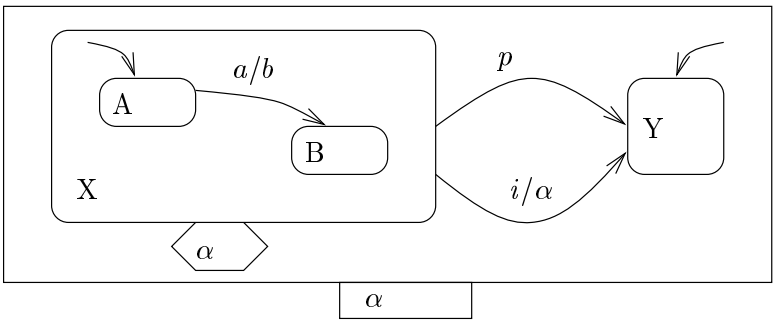


FIG. 2.10 – *Expression des interruptions préemptives*

La figure 2.10 illustre la construction des interruptions préemptives grâce à l'opération qui associe une condition d'activation au processus raffinant. L'interruption par i n'est pas préemptive. Ainsi, si i et a surviennent simultanément, le système passe de l'état A à l'état Y en émettant b. L'interruption par p, en revanche, est préemptive. Si p et a surviennent dans l'état A, le système passe dans l'état Y, mais sans émettre b.

2.4.2 Comparaison avec les Statecharts

ARGOS est bien sûr issu des Statecharts, mais n'en a conservé que la structure hiérarchique des automates. La sémantique construite sur l'hypothèse de synchronisme doit bien plus à ESTEREL qu'aux Statecharts (la description "officielle" de la sémantique implantée dans l'outil Statemate est basée sur une communication synchrone "retardée" : un signal émis par une composante ne peut être utilisé par les autres composantes que lors de la réaction *suivante*. Ce mécanisme est strictement plus faible que la véritable diffusion synchrone, mais beaucoup plus simple à décrire). ARGOS a été construit en dépouillant petit à petit les Statecharts de leurs nombreuses constructions à sémantique ambiguë. Pourtant la plupart de ces constructions peuvent être obtenues en utilisant le jeu d'opérations présenté ici, à condition de disposer de la véritable diffusion synchrone. C'est le cas, très simplement, pour les transitions inter-niveaux sortantes. Dans l'exemple commenté au paragraphe 2.1, la transition issue de l'état On avec la condition mod8 est exactement le codage d'une transition inter-niveaux issue des trois états One, avec la condition c. Toutes les transitions inter-niveaux sortantes des Statecharts (dont il est en général difficile de dire si elles représentent des interruptions préemptives ou non, puisque la définition de leur portée constitue un problème en soi) peuvent ainsi être exprimées par des synchronisations entre les processus raffinants et l'automate contrôleur.

Les transitions entrantes, en revanche, supposent dans le cas général la définition du raffinement avec initialisation du processus raffinant sur les transitions entrantes, qui pose tous les problèmes de ré-incarnation d'ESTEREL. Voir le chapitre 9 pour une discussion de ces aspects.

Le chapitre 3 propose un cadre dépouillé où la comparaison des modes de synchronisation d'ARGOS et des Statecharts est simple.

2.5 A propos de compositionnalité

2.5.1 Propriété de compositionnalité pour ARGOS

Le langage ARGOS est conçu comme un jeu de construction pour fabriquer des objets définis a priori, les machines de Mealy à étiquettes booléennes. Ces objets sont définis de manière syntaxique, mais on se donne une relation d'équivalence qui exprime quelles machines doivent être considérées comme représentant le même comportement réactif. Cette relation est la bisimulation définie plus haut (définition 2.2). On exige que cette relation soit une congruence pour les opérations du langage.

Cette démarche est générale. Quel que soit le type de modèle que l'on désire construire, s'il existe une relation d'équivalence pour déterminer quels modèles représentent en fait le même comportement, malgré leurs différences syntaxiques, cette relation *doit* être une congruence pour les opérateurs du langage. On retrouve ce raisonnement et l'exigence de compositionnalité aux chapitres 3 et 4.

A partir de l'équivalence des modèles, on définit une équivalence de *programmes*, notée $\equiv$, par :

Definition 2.9 : Equivalence de programmes

$$P_1 \equiv P_2 \quad \Longleftrightarrow \quad \begin{cases} \mathcal{S}(P_1) \neq \bot \wedge \mathcal{S}(P_2) \neq \bot \wedge \mathcal{S}(P_1) \approx \mathcal{S}(P_2) & \vee \\ \mathcal{S}(P_1) = \mathcal{S}(P_2) = \bot \end{cases}$$

□

La propriété de *compositionnalité* du langage présenté ici repose sur le théorème suivant :

Théorème 2.1
L'équivalence de programmes est une congruence pour les opérateurs du langage. Autrement dit :

$$\forall P, Q \in \mathcal{P}, \forall \mathcal{C} \text{ contexte d'opérateurs} \quad P \equiv Q \implies \mathcal{C}[P] \equiv \mathcal{C}[Q]$$

□

Si deux programmes ARGOS ont des images bisimilaires par la fonction sémantique, ils se comportent de la même façon dans tout contexte.

2.5.2 Congruences et complète adéquation

La comparaison entre ARGOS et les Statecharts a souvent conduit à un débat entre sémantique opérationnelle+congruence, et sémantique dénotationnelle+propriété de complète adéquation (ou "*full abstraction*"). A ma connaissance, les seules sémantiques à la fois compositionnelles et correctes proposées pour les Statecharts sont dénotationnelles (voir par exemple [HGdR88]). Je

ne me lancerai pas dans une comparaison formelle de ces deux techniques. Notons simplement que la sémantique d'ARGOS présentée ici possède la propriété de complète adéquation, de manière triviale.

En paraphrasant la remarquable introduction de [Sto88], on peut préciser le cadre formel en quatre points (voir figure 2.11) :

- On considère un ensemble $\mathcal{T}$ de *termes*, définis syntaxiquement, et formant un langage. Parmi ces termes, certains, qu'on appellera des *programmes*, correspondent à des comportements observables ; par exemple, dans un langage impératif séquentiel classique, les expressions ne sont pas des programmes, alors que les procédures en sont. Un programme est en général un terme clos, vis-à-vis des identificateurs dont la portée est définie statiquement.

- On définit une notion de *comportement observable* des programmes, par une fonction $\mathcal{O}$ d'un sous-ensemble de $\mathcal{T}$ vers un ensemble de comportements $\mathcal{B}$. Pour un langage déterministe classique, le comportement peut être une fonction des entrées vers les sorties.

- On définit une équivalence de termes par : $P \equiv Q \iff \forall C,\ \mathcal{O}(C[P]) = \mathcal{O}(C[Q])$, où C représente un *contexte* syntaxique définissant un programme. Autrement dit, deux programmes sont dits équivalents s'ils sont interchangeables, dans tout contexte, sans effet sur le comportement global.

- On peut ensuite chercher à caractériser sémantiquement la notion d'équivalence de programmes, en trouvant un ensemble de modèles $\mathcal{M}$ et une fonction sémantique $\mathcal{S}$ de $\mathcal{T}$ vers $\mathcal{M}$ telle que $P \equiv Q$ se ramène à $\mathcal{S}(P) = \mathcal{S}(Q)$. Plus précisément, on dit que le modèle est *correct* lorsque seuls des termes équivalents ont même image par la fonction sémantique, c'est-à-dire $\mathcal{S}(P) = \mathcal{S}(Q) \implies P \equiv Q$. On dit que le modèle est complètement adéquat si, de plus, $P \equiv Q \implies \mathcal{S}(P) = \mathcal{S}(Q)$, c'est-à-dire que deux termes équivalents sont toujours identifiés par la fonction sémantique. D'autre part, en sémantique dénotationnelle, la fonction $\mathcal{S}$ est décrite fonctionnellement, et la substitutivité garantit que l'égalité des modèles est une congruence pour les opérations du langage.

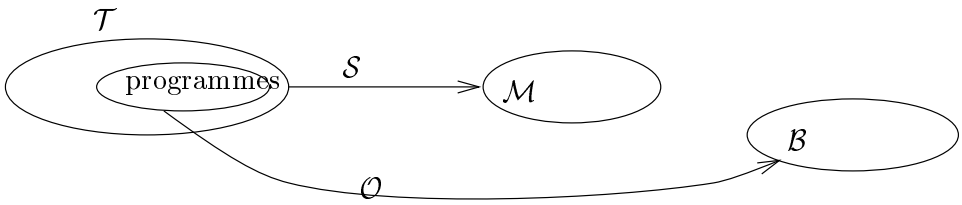


FIG. 2.11 – *Termes, programmes, comportements et modèles*

Dans le cas d'ARGOS, tous les termes sont des programmes, puisque la plus petite structure syntaxique est l'automate, et coïncide avec la notion de comportement observable. On a donc aussi $\mathcal{B} = \mathcal{M}$ et $\mathcal{S} = \mathcal{O}$. La propriété de congruence énoncée plus haut garantit donc $P \equiv Q \iff \mathcal{S}(P) = \mathcal{S}(Q)$.

Chapitre 3

Modèles généraux pour les langages synchrones

Sources : *Ce chapitre est tiré pour l'essentiel de [JM94b]. La comparaison des modes de synchronisation d'*ARGOS *et des Statecharts n'a jamais été publiée telle que.*

Le travail publié dans [JM94b], qui peut être vu comme un certain aboutissement des multiples ré-écritures de la sémantique d'ARGOS, a son origine dans de nombreuses et intéressantes discussions avec Amir Pnueli à Rehovot en 1991, au cours desquelles nous tentions de rapprocher la sémantique d'ARGOS d'une version de la sémantique des Statecharts (publiée dans [PS88]). Cet aspect du travail est encore d'actualité car, à ma connaissance, les évolutions récentes de la sémantique des Statecharts suivent les idées présentées dans [PS88] (bien que la sémantique officiellement implantée dans l'outil Statemate soit basée sur une communication synchrone retardée).

Autour du même contenu technique, les arguments ont évolué au cours de 4 ou 5 présentations à des publics plus ou moins familiers des langages synchrones. Je donne ici la structure de l'article, en complétant l'exposé des motivations. Les exposés faisaient une large part à des arguments de *compositionnalité*, que j'ai par ailleurs résumés au chapitre 2, section 2.5.2. En revanche, on oubliera avec profit la conclusion hâtive de l'article, qui tentait de rapprocher un critère esthétique essentiellement subjectif de "simplicité" d'un mode de communication synchrone, d'un critère objectif de quantité d'information nécessaire à assurer la congruence. Reste un cadre unifié permettant d'exprimer divers modes de communication synchrone.

Notons enfin que l'on retrouve dans le chapitre 14, à propos d'algorithmique en DEUG et de conception des itérations, l'essentiel du discours sur la définition de congruences par le biais de fonctions d'abstraction.

3.1 Motivations

La plus grande difficulté rencontrée, depuis le début, dans les tentatives de comparaison d'ARGOS— c'est-à-dire de la stricte application de l'hypothèse de synchronisme à des compositions d'automates — et des Statecharts, réside dans le caractère non compositionnel de la plupart des sémantiques proposées pour les Statecharts (à l'exception d'une sémantique dénotationnelle proposée dans [HGdR88], et de la sémantique en algèbre de processus de [US94], laquelle présente par ailleurs quelques erreurs de fond).

En plongeant la sémantique des réactions en chaîne dans le modèle général proposé ici, on fournit, par effet de bord, une sémantique compositionnelle pour un sous-ensemble du langage présenté dans [PS88]. On suggère par ailleurs comment atteindre le langage complet. Cela permet de comparer ARGOS et cette version des Statecharts en se concentrant sur la sémantique de la communication entre composants, et en s'abstrayant, temporairement au moins, des constructions comme le raffinement, avec son cortège de problèmes liés à la présence des transitions inter-niveaux.

3.2 Cadre général

Le cadre général que nous proposons ici doit permettre de définir la sémantique d'un mode de communication synchrone avec un minimum d'efforts, simplement en instanciant les quelques paramètres qui apparaissent dans les règles de sémantique. On peut ainsi comparer plusieurs modes de communication en comparant les paramètres utilisés. Le cadre général garantit également l'existence d'une congruence.

Il existe de nombreux formats généraux pour décrire des compositions parallèles et des modes de communication, par exemple [Arn92, AB84]. Le but n'est pas de comparer un nouveau format avec ceux-ci, sur des critères d'expressivité ou d'existence d'une axiomatisation par exemple.

Le format proposé est une variante adaptée à la recherche de congruences, défini en suivant deux idées. Tout d'abord, c'est un format d'algèbre de processus, mais les objets élémentaires sont des automates. Cela évite de considérer un opérateur de récursion, puisque la récursion n'est utilisée, en pratique, que pour décrire des automates à partir du préfixage et du choix non déterministe. En général il faut d'ailleurs résoudre les problèmes d'imbrication de la récursion et de la composition parallèle, qui produisent des objets non réguliers. D'autre part, dans la plupart des algèbres de processus, la sémantique du mode de communication est répartie entre une opération de produit — ou mise en parallèle —, et une opération unaire d'encapsulation — ou restriction, ou encore masquage. En CCS [Mil80] par exemple, les actions a et $\overline{a}$ se fondent en une action invisible τ, lors de la composition parallèle. L'opération de restriction est utilisée pour forcer la synchronisation entre deux composants (sur leurs actions communes) en supprimant, dans leur composition parallèle, les transitions étiquetées par a ou $\overline{a}$ seul. Pour des modes de communication plus complexes, il est parfois difficile de séparer ce qui doit être exprimé dans la mise en parallèle, et ce qui doit être exprimé dans une loi d'encapsulation.

Dans le modèle proposé ici, la mise en parallèle est un simple produit qui conserve l'information maximale. La sémantique de la communication est exprimée entièrement dans la loi de l'encapsulation. Cela permet, en particulier, d'inventer d'autres opérations de composition, par exemple des opérations asymétriques comme le raffinement d'ARGOS, sans remettre en cause la communication entre composants.

3.2.1 Syntaxe et sémantique

On définit un ensemble $\mathcal{L}ab$ d'*étiquettes*, aussi bien pour les systèmes de transitions qui constituent les objets de base des programmes, que pour leurs modèles. Une *étiquette* est un multi-ensemble d'objets qui peuvent être atomiques ou eux-mêmes structurés, selon la complexité du mode de communication à décrire. Un système de transitions est un n-uplet $LTS = (Q, q_0, T)$ (ensemble d'états, état initial et transitions) et les transitions sont étiquetées par des éléments de $\mathcal{L}ab$ (c'est-à-dire $T \subseteq Q \times \mathcal{L}ab \times Q$).

L'ensemble $\mathcal{P}$ des programmes est décrit par la grammaire : $P ::= LTS \mid P \| P \mid \overline{P^Y}$, ou $\|$ est la composition parallèle et $\overline{P^Y}$ l'encapsulation. Noter que le paramètre de cette opération est en relation avec la nature de $\mathcal{L}ab$. Ce peut être un élément, un sous-ensemble ou une sous-structure de $\mathcal{L}ab$, si celui-ci est structuré.

La sémantique est une fonction $\mathcal{S}$ de l'ensemble $\mathcal{P}$ des programmes vers l'ensemble des systèmes de transitions étiquetées décrit ci-dessus. Pour les composants de base, on a simplement : $\mathcal{S}(LTS) = LTS$.

Dans une composition parallèle *synchrone*, une réaction du programme global est toujours constituée d'une réaction de chacun des composants. Notons $\mathcal{S}(P_i) = (Q_i, q_{0i}, T_i)$ le système de transitions associé au composant P_i ($i \in \{1, 2\}$). $\mathcal{S}(P_1 \| P_2)$ est alors de la forme $(Q_1 \times Q_2, q_{01} \| q_{02}, T')$, et T' est donné par :

$$\frac{q_1 \xrightarrow{L_1} q_1', \quad q_2 \xrightarrow{L_2} q_2'}{q_1 \| q_2 \xrightarrow{L_1 \uplus L_2} q_1' \| q_2'} \qquad [\mathrm{P}]$$

$\uplus$ dénote l'union de multi-ensembles, qui préserve les doublons. La composition parallèle ainsi décrite préserve l'information maximale sur les deux transitions qui participent à une réaction globale du système, à condition toutefois d'accepter l'hypothèse raisonnable selon laquelle la composition parallèle est à la fois commutative et associative. Dans le cas contraire, il faudrait conserver, au lieu de l'union des multi-ensembles, un vecteur d'étiquettes. C'est le cas dans le modèle de [Arn92] ; en pratique, dans le système MEC [ABC94], on peut construire explicitement la commutativité et l'associativité de la mise en parallèle, en respectant certaines contraintes sur les vecteurs de synchronisation.

Pour l'encapsulation, on utilise un prédicat $\mathcal{L}$ et une fonction de masquage $\mathcal{H}$. Notons $\mathcal{S}(P) = (Q, q_0, T)$ le système de transitions associé à P. Le système associé à $\overline{P^Y}$ est de la forme $(\overline{Q^Y}, \overline{q_0{}^Y}, T')$ où $\overline{Q^Y} = \{\overline{q^Y}, q \in Q\}$ et T' est défini par :

$$\frac{q \xrightarrow{L} q', \quad \mathcal{L}_Y(L)}{\overline{q^Y} \xrightarrow{\mathcal{H}_Y(L)} \overline{q'^Y}} \qquad [\mathrm{E}]$$

$\mathcal{L}$ est le *critère local*. C'est un prédicat qui détermine quelles transitions du modèle de P persistent dans le modèle du programme encapsulé. Ce prédicat ne porte que sur l'étiquette de la transition, indépendamment des autres transitions, d'où son nom de critère *local*. Une transition dont l'étiquette satisfait au critère est conservée ; son étiquette est modifiée par la fonction $\mathcal{H}$ qui permet de cacher ce qui concerne le paramètre de l'encapsulation Y.

3.2.2 Equivalences, congruences et compositionnalité

Nous définissons une bisimulation paramétrée par une équivalence d'étiquettes notée $\mathcal{R}_L$. L'idée est de montrer que cette bisimulation est une congruence, pourvu que l'équivalence d'étiquettes soit une congruence pour les opérateurs qui portent sur les étiquettes, c'est-à-dire l'union de multi-ensembles $\uplus$ et l'opération de masquage $\mathcal{H}_Y$.

Definition 3.1 : Bisimulation paramétrée
Notons $(Q_1, q_{01}, T_1) \sim_L (Q_2, q_{02}, T_2)$ la bisimulation de deux systèmes de transitions, paramétrée par $\mathcal{R}_L$.

$(Q_1, q_{01}, T_1) \sim_L (Q_2, q_{02}, T_2)$ si et seulement si il existe une relation $\mathcal{R} \subseteq Q_1 \times Q_2$ telle

que :

$$(q_{01}, q_{02}) \in \mathcal{R} \text{ et}$$

$$(q_1, q_2) \in \mathcal{R} \implies \begin{cases} q_1 \xrightarrow{L_1} q_1' \implies \exists q_2', L_2 \text{ t.q.} \begin{cases} q_2 \xrightarrow{L_2} q_2' \\ (q_1', q_2') \in \mathcal{R} \\ \mathcal{R}_L(L_1, L_2) \end{cases} \\ q_2 \xrightarrow{L_2} q_2' \implies \exists q_1', L_1 \text{ t.q.} \begin{cases} q_1 \xrightarrow{L_1} q_1' \\ (q_1', q_2') \in \mathcal{R} \\ \mathcal{R}_L(L_1, L_2) \end{cases} \end{cases}$$

□

Cette définition induit une équivalence de programmes : $P_1 \equiv_L P_2 \iff \mathcal{S}(P_1) \sim_L \mathcal{S}(P_2)$. On peut alors prouver, par une simple induction sur la structure des programmes, que : *Si* $\mathcal{R}_L$ *preserve le prédicat* $\mathcal{L}$ *et est préservée par* $\mathcal{H}_Y()$ *et* $\mathbb{U}$*, alors* $\equiv_L$ *est une congruence.*

Le point intéressant de ce cadre de travail est qu'il permet d'étudier des congruences de programmes en se ramenant à l'étude de congruences d'étiquettes. D'autre part, une équivalence sur un ensemble E peut être définie par l'intermédiaire d'une fonction f non injective de E vers F, en écrivant $e_1 \sim e_2 \iff f(e_1) = f(e_2)$. On se ramène donc à l'étude de telles fonctions non injectives, qu'on appelle des *abstractions.*

L'égalité d'étiquettes est la congruence la plus fine. La relation la plus intéressante est l'équivalence la plus grossière qui préserve le critère local et est préservée par les opérations sur les étiquettes. Toutefois il peut se révéler difficile de définir cette équivalence intéressante de manière constructive.

On peut essayer de l'atteindre par des affaiblissements de l'égalité, au moyen de fonctions d'abstraction Ab :

Ab : ensemble d'étiquettes $\longrightarrow$ ensemble d'étiquettes abstraites
$\mathcal{R}_L(L_1, L_2)$ si et seulement si $Ab(L_1) = Ab(L_2)$.

Une fois choisie une fonction d'abstraction, il reste à vérifier que l'équivalence correspondante est bien une congruence. Il est ensuite possible d'intégrer la fonction d'abstraction dans les règles de sémantique, en construisant directement les étiquettes abstraites (la congruence assure que les fonctions de test et les opérations de combinaison d'étiquettes ne s'appuient justement pas sur l'information éliminée lors de l'astraction). On utilise alors une bisimulation classique, sans paramètre.

3.3 Applications

L'article présente comment exprimer trois calculs dans le cadre général défini plus haut.

3.3.1 Le calcul synchrone de Milner

Nous reprenons ici le cas du calcul synchrone de [Mil89]. Le calcul présenté en est une restriction au cas où les objets de base sont des automates.

Dans le calcul de [Mil89], les objets atomiques sont appelés *particules.* On utilise donc un ensemble de particules $\mathcal{P}ar = \{a, b, ...\}$. Un ensemble d'*actions* est défini comme un groupe commutatif ($\mathcal{A}ct$, 1, $\times$, -). 1 est l'action d'attente, $\times$ dénote l'occurrence simultanée de deux actions et la barre dénote la complémentation d'action : $a \times \overline{a} = 1$. L'ensemble des actions est

engendré par l'ensemle de particules. Une action peut toujours être mise sous la forme $a_1^{z_1}...a_n^{z_n}$, où les a_i sont des particules distinctes, et les z_i sont des entiers relatifs. L'opération de restriction notée $P \uparrow Y$ est paramétrée par un sous-groupe Y de $\mathcal{A}ct$, et elle supprime toutes les transitions dont l'étiquette n'appartient pas à Y.

On plonge ce calcul dans le cadre général proposé ici en définissant les objets élémentaires comme l'ensemble des particules et complémentations de particules et les étiquettes comme des multi-ensembles d'objets de cette forme. L'opération binaire $\times$ de composition d'actions étant commutative et associative, on l'étend à un nombre quelconque d'arguments :

$$\begin{array}{l}\bigotimes : \text{un multi-ensemble d'actions } \mapsto \text{une action}\\ \bigotimes(\emptyset) = 1\\ \bigotimes(\{\!| e |\!\} \uplus E) = \bigotimes(E) \times e\end{array}$$

Le critère local et la fonction de masquage sont définis par :

$$\mathcal{L}_Y(L) = \bigotimes(L) \in Y, \quad \mathcal{H}_Y(L) = \bigotimes(L)$$

En abstrayant les étiquettes selon $Ab(L) = \bigotimes(L)$, on retrouve la sémantique usuelle du calcul de [Mil89]. En effet, une action peut être vue comme l'astraction d'un multi-ensemble de particules et complémentations de particules, en retenant seulement la différence entre le nombre de a_i et le nombre de $\overline{a_i}$, pour chaque a_i.

3.3.2 Diffusion synchrone

L'ensemble des particules est l'ensemble des *signaux* d'entrée/sortie : $\mathcal{P}ar = \{a, b, c...\}$. Une *réaction* est de la forme $\alpha =< \alpha^+, \alpha^-, \alpha^o >$, où α^+, α^- et α^o sont des sous-ensembles de $\mathcal{P}ar$, pas nécessairement disjoints. α^+, α^- donnent la condition d'entrée (le monôme dans lequel les éléments de α^+ sont positifs et ceux de α^- négatifs), α^o donne l'ensemble des signaux émis. Les *étiquettes* sont des multi-ensembles de réactions.

Pour une étiquette L, on définit $I^+(L)$ (entrées positives), $I^-(L)$ (entrées négatives) et $O(L)$ (sorties) :

$$I^+(L) = \bigcup_{<\alpha^+,\alpha^-,\alpha^o>\in L} \alpha^+, \quad I^-(L) = \bigcup_{<\alpha^+,\alpha^-,\alpha^o>\in L} \alpha^-, \quad O(L) = \bigcup_{<\alpha^+,\alpha^-,\alpha^o>\in L} \alpha^o$$

L'opération d'encapsulation est définie en instanciant le critère $\mathcal{L}_Y(L)$ et la fonction $\mathcal{H}_Y(L)$ par :

$$\begin{array}{l}\mathcal{L}_Y(L) = (I^+(L) \cap Y \subseteq O(L) \ \wedge \ I^-(L) \cap Y \cap O(L) = \emptyset)\\ \mathcal{H}_Y(L) = \{\!| \ \alpha[Y], \alpha \in L |\!\}\end{array}$$

où $\alpha[Y]$ est la restriction de α aux signaux (particules) de Y : $< \alpha^+, \alpha^-, \alpha^o > [Y] =< \alpha^+ - Y, \alpha^- - Y, \alpha^o - Y >$.
Un rapide coup d'oeil aux définitions du critère local et de la fonction de masquage montre que les étiquettes peuvent être abstraites par la fonction :

$$Ab(L) = (I^+(L), I^-(L), O(L))$$

qui induit une congruence. On retrouve ainsi la sémantique présentée au chapitre 2, sans les aspects de détection des programmes incorrects.

3.3.3 Réactions en chaîne

On utilise les mêmes ensembles de particules et d'étiquettes que pour la diffusion synchrone décrite ci-dessus. Pour une étiquette L, on définit également $I^+(L)$, $I^-(L)$ et $O(L)$. Les réactions en chaîne diffèrent de la diffusion synchrone dans l'expression du critère local et de la fonction de masquage :

$$\begin{aligned}
\mathcal{L}_Y(L) &= (1) \quad (I^+(L) \cap Y \subseteq O(L) \wedge I^-(L) \cap Y \cap O(L) = \emptyset) \quad \wedge \\
&\quad (2) \quad \exists f : L \longrightarrow [1, |L|],\ f \text{ bijective, telle que :} \\
&\quad \forall n \in [1, |L|], I^+(f^{-1}(n)) \cap Y \subseteq \bigcup_{i=1}^{n-1} O(f^{-1}(i)) \\
\mathcal{H}_Y(L) &= \{\!| \ \alpha[Y], \alpha \in L |\!\}
\end{aligned}$$

La condition (2) exprime que l'étiquette L peut être *totalement ordonnée* (d'après la fonction de numérotation f) de telle manière que, dans toute réaction, les signaux supposés présents ont été effectivement émis par des réactions *précédentes*. Il faut donc commencer par des réactions qui sont rendues possibles en connaissant seulement le statut des signaux d'entrée.

On peut abstraire quelque peu les étiquettes, en remarquant que l'existence d'un ordre sur les diverses réactions qui composent une étiquette ne dépend pas de la partie négative des étiquettes. On écrit donc :

$$Ab(L) = (I^-(L), \{\!| < \alpha^+, \emptyset, \alpha^o > | \ \exists \alpha^- \ < \alpha^+, \alpha^-, \alpha^o > \in L |\!\})$$

Il faut prouver que cette abstraction induit une congruence, c'est-à-dire que :

$$Ab(L_1) = Ab(L_2) \quad \Longrightarrow \quad \forall L, \forall Y, \begin{cases} Ab(L_1 \uplus L) = Ab(L_2 \uplus L) \\ Ab(\mathcal{H}_Y(L_1)) = Ab(\mathcal{H}_Y(L_2)) \\ \mathcal{L}_Y(L_1) = \mathcal{L}_Y(L_2) \end{cases}$$

Pour l'union de multi-ensembles et la fonction de masquage, c'est assez simple : la propriété vient de la distributivité de I^+, I^- et O sur les opérations d'union et de restriction. Le premier point de la définition du critère local est également simple à prouver. La difficulté vient du deuxième point : l'existence d'un ordre total sur les réactions qui composent une étiquette. C'est possible, toutefois, et on obtient donc une première abstraction.

Peut-on aller plus loin ? Cette première abstraction isole l'ensemble des signaux qui doivent être absents dans une étiquette, mais conserve toujours l'information maximale sur les autres. En particulier, on a encore une réaction par composant parallèle de base du programme. Or le mécanisme des réactions en chaîne induit une mise en parallèle *idempotente*. Une première simplification pourrait donc être de traiter des ensembles, et non des multi-ensembles. D'autre part, considérons une étiquette qui contient les deux réactions positives (i^+, o_1) et (i^+, o_2). En ce qui concerne l'existence d'un ordre, cette étiquette se comporte comme : $(i^+, o_1 \cup o_2)$. On peut imaginer quelques autres simplifications de ce style. On retrouve ici la suggestion d'Amir Pnueli d'axiomatiser l'équivalence d'étiquettes recherchée, en écrivant des axiomes du genre : $\{(i^+, o_1), (i^+, o_2)\} = \{(i^+, o_1 \cup o_2)\}$.

3.4 Commentaires et réflexions

3.4.1 Comparaison ARGOS/Statecharts

L'expression, dans notre cadre, du mode de communication des Statecharts, caractérisé par une notion de réaction en chaîne entre composants, fournit une version compositionnelle de la

sémantique présentée dans [PS88]. L'étude de quelques abstractions d'étiquettes permet d'autre part de mieux cerner la sémantique des réactions en chaîne. La proposition d'axiomatiser l'équivalence d'étiquettes est une réminiscence des discussions avec Amir Pnueli.

L'exemple du dialogue instantané illustre bien la différence entre les divers modes de communication d'ARGOS (la diffusion synchrone exposée au chapitre 2 et reprise ci-dessus) et des Statecharts (la version compositionnelle présentée ici).

Exemple 3.1 : Dialogue instantané
La figure 3.1 donne deux programmes ARGOS congruents qui représentent un dialogue instantané entre deux composants : en réaction à une cause extérieure C, le premier composant tente de changer d'état (de A à B), à condition que l'autre composant soit dans l'état ON. Pour cela, il émet la question Q, et ne change d'état que si le deuxième composant répond Y. La version (**a**) décrit le premier composant par deux systèmes. Dans la version (**b**), le premier composant est directement donné par un automate.

Les deux versions du premier composant sont congruentes, puisque la deuxième est la evrsion compilée de la première.

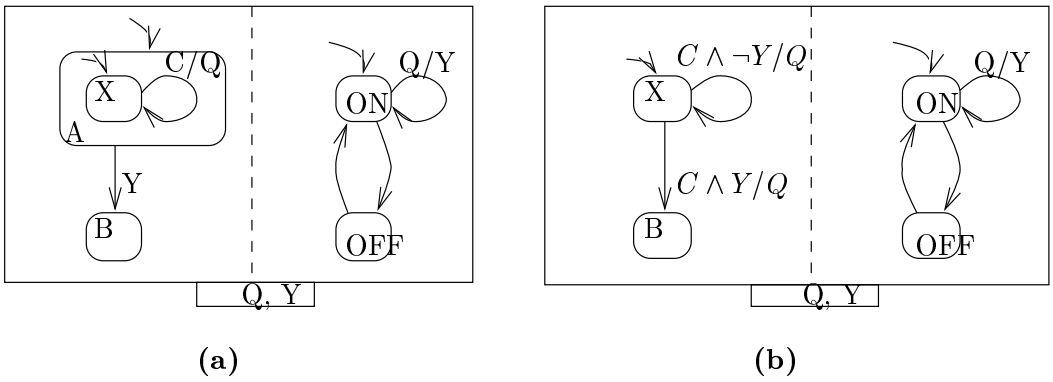


FIG. 3.1 – *Deux versions du dialogue instantané*

Dans la sémantique des Statecharts présentée ci-dessus, le programme (**a**) a le comportement attendu, alors que le deuxième n'a pas de transition pour l'événement extérieur réduit au signal C. En effet, il est impossible de trouver une chaîne de transitions élémentaires des automates, pour expliquer une réaction globale de X vers B. □

Cet exemple conduit à s'interroger sur la notion de *causalité* adéquate pour ARGOS. La discussion apparaît au chapitre 9, paragraphe 9.6 qui propose un langage mixte ARGOS+ESTEREL.

3.4.2 Extensions du modèle

Nous proposions l'extension du modèle général au cas de plusieurs opérations de produit, éventuellement asymétriques, et la prise en compte d'équivalences de systèmes de transitions dans lesquelles les états peuvent également porter des informations pertinentes (et non plus seulement les transitions). La première de ces extensions permet de coder le raffinement d'ARGOS, sans remettre en cause l'expression de la diffusion synchrone entre composants ; la deuxième extension permet d'exprimer, par exemple, la détection des programmes causalement incorrects, puisqu'il

est possible d'associer à un état une information *globale* sur l'ensemble des transitions qui en sont issues.

Lors de la présentation de ce travail au groupe de travail sur les langages synchrones (Dagstuhl, novembre 1994), Albert Benveniste suggéra que la sémantique de Signal pouvait également être exprimée dans ce modèle, en utilisant des ordres partiels comme étiquettes de transitions.

Chapitre 4

Non déterminisme et langages synchrones

Source : *ce chapitre est tiré de [MH96], que nous joignons en annexe. Le détail des preuves n'est pas repris ici. Il me paraît préférable de consacrer ce chapitre à l'exposé des motivations et des problèmes posés.*

On fait souvent la publicité des langages synchrones en insistant sur le fait qu'ils ont réconcilié *concurrence* et *déterminisme*. Ils offrent une vue logique de la concurrence, comme notion de structuration de programmes, un mode de synchronisation puissant, et tout cela en garantissant que la composition d'objets déterministes donne un résultat également déterministe. C'est une caractéristique importante lorsque le but est de *programmer* des systèmes réactifs.

Cela peut toutefois donner l'impression que les langages synchrones sont condamnés au déterminisme. Or le déterminisme n'est pas toujours une propriété souhaitable. En particulier, si le langage est utilisé pour *spécifier* plutôt que pour programmer, une spécification abstraite nécessite en général des objets non déterministes. C'est un moyen simple de décrire un *ensemble* de comportements de manière concise.

Un des effets de bord de l'étude présentée ici est de montrer que les langages synchrones ne sont pas condamnés au déterminisme et peuvent, si on propose une extension raisonnable, être utilisés comme langages de spécification.

Une autre conséquence est la mise en évidence d'une distinction entre non déterminisme intrinsèque, maîtrisable, voulu par le programmeur, et non déterminisme apparaissant dans les résultats de compositions parallèles pour des raisons plus ou moins mystérieuses selon le degré de difficulté de la sémantique du langage. Les difficultés techniques rencontrées dans ce travail n'étant dues qu'au respect de cette distinction, nous commençons par motiver l'approche suivie et les choix effectués.

Enfin ce travail a conduit à s'interroger sur ce que représente le non déterminisme dans la description de systèmes informatiques, une fois admise l'idée que l'expression *programme non déterministe* constitue un oxymoron (c'est bien parce que les programmes sont toujours déterministes qu'il est si difficile de *programmer* un générateur aléatoire). Considérer que les programmes sont toujours déterministes n'est pas en contradiction avec l'existence de *spécifications* non déterministes, au sens propre du terme, c'est-à-dire *insuffisamment déterminées*. La sémantique collatérale des opérateurs booléens dans ADA par exemple, qui constitue une spécification du compilateur, est bien non déterministe, puisqu'elle ne détermine pas l'ordre dans lequel doivent

être évalués les opérandes. Les compilateurs ADA et les codes compilés sont toujours déterministes sur ce point, au moins pour des programmes séquentiels (pour les programmes parallèles, le non déterminisme apparent — c'est-à-dire la non reproductibilité des exécutions — est dû à une connaissance incomplète du système à l'exécution, et c'est précisément cet inconvénient que les langages synchrones se sont attachés à résoudre).

Techniquement, nous recherchons une équivalence des machines de Mealy booléennes non déterministes, compatible avec l'idée selon laquelle un objet non déterministe représente un *ensemble* d'objets déterministes.

4.1 Motivations

Les motivations sont de deux ordres. Elles visent à résoudre aussi bien les problèmes dans les cas où : 1) le but est de décrire des objets non déterministes (par exemple parce qu'ils représentent de manière concise l'ensemble des comportements de l'environnement du système) ; 2) le but est de décrire des objets déterministes, mais dont l'abstraction (par exemple booléenne) manipulable par les algorithmes et outils associés au langage est, elle, non déterministe.

Le deuxième cas est apparu lorsque nous avons voulu décorer les transitions des automates de programmes ARGOS par des informations sur des variables continues, afin de décrire des systèmes temporisés ou hybrides (cf. pour plus de détails les chapitres 10 et 11).

Nous détaillons tout d'abord les deux types de cas à résoudre.

4.1.1 Représentation d'un ensemble de comportements modélisation de l'environnement

Dans la spécification des protocoles de communication, un cas typique d'utilisation du non déterminisme est constitué par la modélisation d'une ligne de transmission non fiable, qui peut transmettre les messages correctement, les perdre ou les réordonner...

Lorsqu'on introduit dans le système un composant non déterministe pour représenter la ligne de transmission, il s'agit bien de représenter de manière concise tout l'ensemble infini des comportements possibles de la ligne qui, à chaque transmission de message, "choisit" arbitrairement de transmettre correctement, perdre ou réordonner les messages. Lorsqu'on prouve des propriétés de fonctionnement du protocole pour un tel système, on a bien prouvé quelque chose, globalement, pour tous les comportements possibles de la ligne.

Ce cas de figure se présente couramment dès qu'il est nécessaire d'intégrer une description de l'environnement d'un système, aux fins de vérification de propriétés de fonctionnement de l'ensemble. Savoir si l'environnement à décrire peut ou non être représenté par un processus complètement déterministe, pour peu que l'on connaisse les lois physiques appropriées et les conditions initiales, constitue un problème philosophique certes intéressant, mais dont l'étude dépasse le cadre de ce document et les compétences de son auteur. Quoi qu'il en soit, la modélisation détaillée et déterministe du comportement d'un environnement physique est, d'une part hors de portée de moyens informatiques, d'autre part, on l'espère, inutile. On se contente donc d'une abstraction maîtrisable, en général non déterministe, ce qui rejoint, en quelque sorte, le deuxième cas.

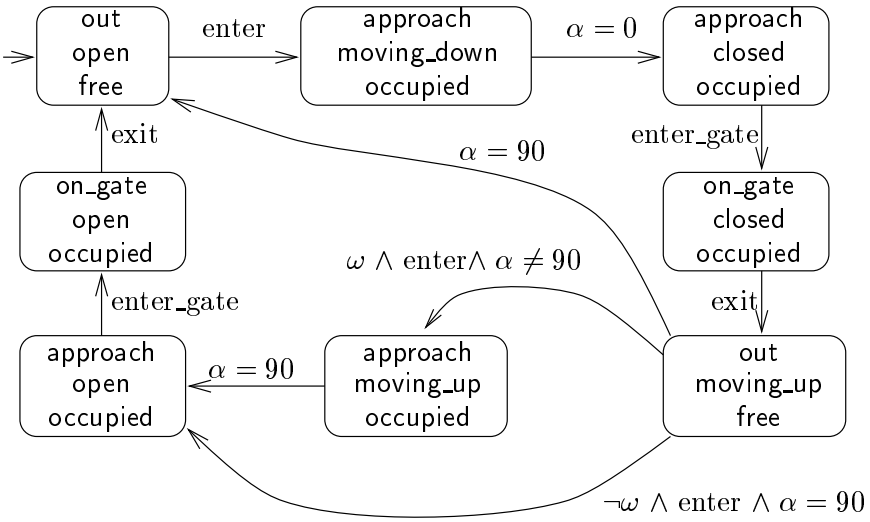


FIG. 4.1 – *Déterminisation par un signal auxiliaire ω*

4.1.2 Abstraction d'un comportement déterministe

Le deuxième cas courant d'apparition d'objets non déterministes est plus "technique". Il est dû aux limitations des algorithmes et outils associés au langage. On veut par exemple décrire un comportement intrinsèquement déterministe, mais pour lequel le déterminisme est assuré par des informations non traitées par le compilateur. Ces informations ne sont toutefois pas inutiles. Elles peuvent être transmises à un outil plus puissant qui saura les analyser.

C'est le cas rencontré dans le couplage ARGOS+POLKA.

Exemple 4.1 : Traitement du non déterminisme dans ARGOS+POLKA
L'automate ci-dessous décrit le comportement global d'un passage à niveau. Les états sont des triplets d'états des trois composants : le train, qui peut être dans la zone protégée par la barrière du passage à niveau ou en dehors (*out*, *approach*, *on_gate*) ; la barrière, qui peut être ouverte, fermée ou en mouvement (*open*, *closed*, *moving_down*, *moving_up*) ; la section de voie du passage à niveau, qui peut être occupée ou libre (*occupied*, *free*). La variable α décrit l'angle que fait la barrière avec le sol, en degrés.

Cet automate peut être obtenu par compilation d'un programme ARGOS à trois composants parallèles, après élimination des transitions non tirables par l'outil POLKA. Cette méthode fait l'objet du chapitre 11, où nous décrivons cet exemple en détail.

Considérons ici que l'automate est décrit directement. Un cas de non déterminisme apparent est dû aux transitions déclenchées par le signal *enter* (le train entre dans la zone protégée, et émet ce signal en passant une pédale sur la voie), à partir de l'état global *out/moving_up/free*. En effet, la réaction dépend de l'état de la barrière à ce moment-là, c'est-à-dire de la valeur de l'angle α. On trouve donc les deux conditions $\alpha = 90 \wedge enter$ et $\alpha \neq 90 \wedge$ *enter*.

□

On résout le problème en ajoutant un signal ω qui permet d'ecrire deux conditions booléennes disjointes : *enter*$\wedge\omega$ et *enter* $\wedge\neg\omega$.

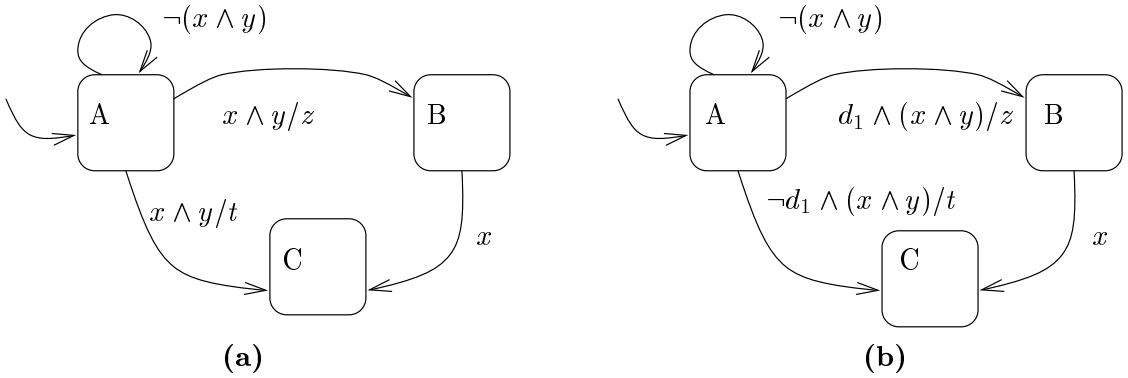


FIG. 4.2 – *Introduction d'oracles — cas simple*

4.1.3 La technique des oracles

La communauté des utilisateurs de langages synchrones n'ayant pas attendu cette étude pour utiliser ces langages comme des langages de spécification, il existe depuis longtemps une astuce très simple pour représenter le non déterminisme dans un langage synchrone.

Puisque l'on décrit naturellement des systèmes à entrées/sorties, il suffit d'introduire des entrées supplémentaires, que l'on baptise *oracles*, ce qui déplace le non déterminisme à l'extérieur du système, en en rendant responsable une entité douée de la capacité de choisir, que nous appellerons par la suite *démon*[1].

Nous illustrons tout d'abord par des exemples en ARGOS la technique d'introduction des oracles et les problèmes qui se posent.

Exemple 4.2
Dans l'exemple donné Figure 4.2, un seul cas de non déterminisme apparaît, et il est en outre binaire. L'introduction d'une entrée oracle d_1 suffit à résoudre le non déterminisme.

□

La situation se complique si les cas de non déterminisme ne sont plus binaires, mais par exemple ternaires. Il faut alors introduire plusieurs oracles booléens.

Exemple 4.3
Dans l'exemple donné Figure 4.3, un cas de non déterminisme ternaire apparaît. On introduit deux oracles d_1 et d_2.

(c) constitue une solution correcte, alors que **(b)** est incorrecte. En effet, en introduisant les conditions $d_1 \wedge d_2$, $d_1 \wedge \neg d_2$ et $\neg d_1 \wedge d_2$, on a transformé un système réactif non déterministe en système déterministe mais *non réactif*. □

Exemple 4.4

1. Le choix du terme "démon" est discutable. La communauté de recherche sur les spécifications relationnelles utilise les termes de non-déterminisme "*angélique*" ou "*démoniaque*" selon que l'entité extérieure qui résoud les choix non déterministes a tendance à favoriser, ou au contraire à contrarier, la réalisation des propriétés attendues. Cette distinction n'a toutefois pas de sens dans notre contexte. Il fallait donc choisir entre ange et démon...

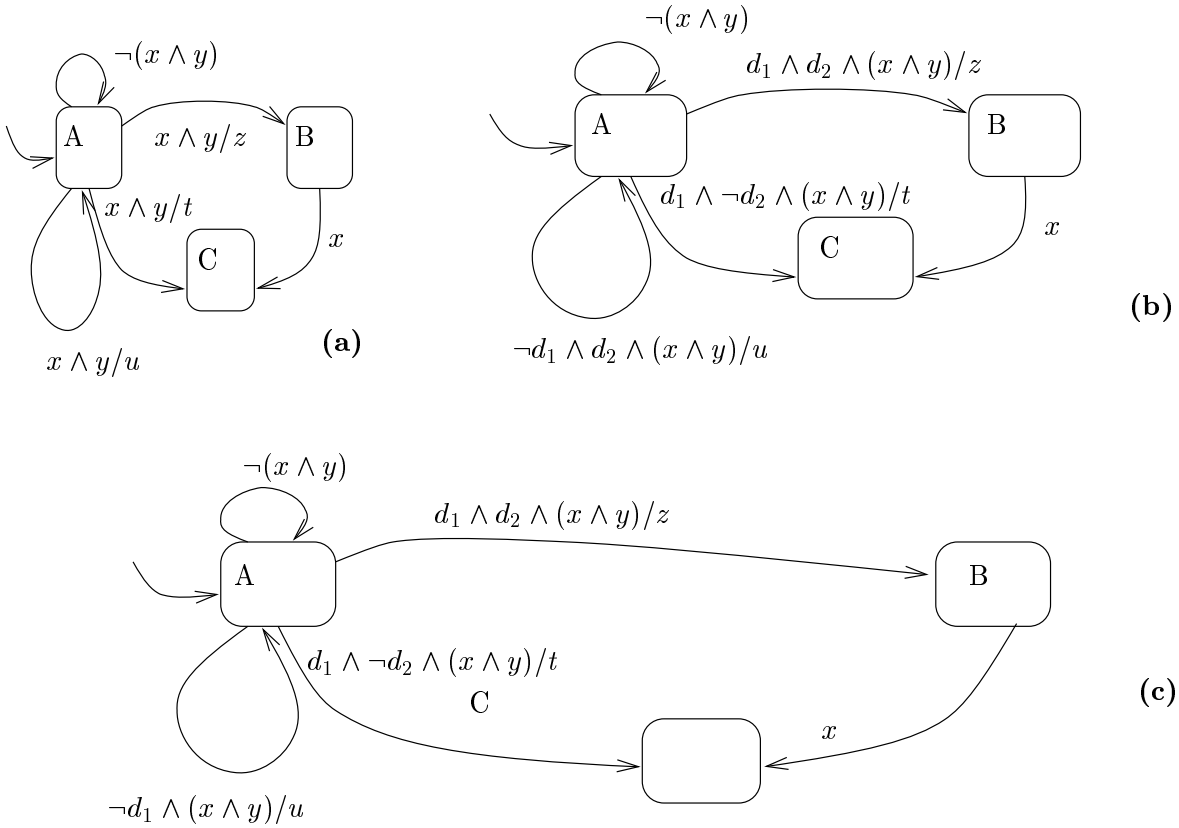


FIG. 4.3 – *Introduction d'oracles — cas de non déterminisme ternaire*

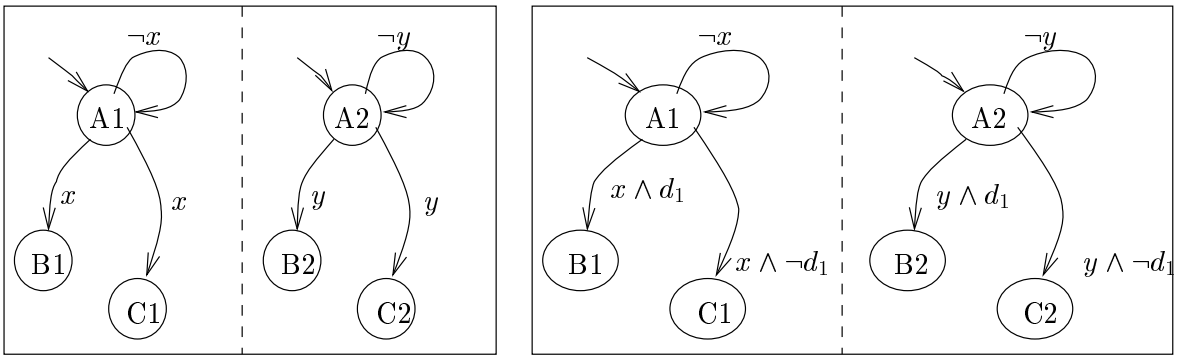


FIG. 4.4 – *Introduction d'oracles — interférence avec les constructions du langage*

Dans l'exemple donné Figure 4.4, on montre un mauvais choix pour l'introduction d'oracles dans un système parallèle dont les deux composants présentent des cas de non déterminisme binaires.

Utiliser le même oracle pour résoudre les cas de non déterminisme dans les deux composants parallèles revient à imposer une synchronisation entre ces composants : dans le mode de communication par diffusion synchrone, les composants parallèles sont implicitement synchronisés sur leurs entrées communes. Dans le système déterminisé, la configuration d'états $A'B''$ est inaccessible depuis AB, puisqu'il faudrait exécuter simultanément une transition où d_1 est vrai, et une transition où il est faux. On se convainc facilement que toute composition parallèle impose l'utilisation de deux ensembles d'oracles disjoints. □

4.2 Formalisation de la technique des oracles

4.2.1 Schéma de principe et propriétés recherchées

Quelle que soit la motivation pour l'introduction d'entrées oracles dans un programme, on peut procéder de manière structurelle. Les composants parallèles peuvent être déterminisés indépendamment les uns des autres, pour peu que l'on choisisse des ensembles d'oracles disjoints. L'opération de raffinement permet une réutilisation partielle des entrées oracles puisque les périodes d'activation des différents comportements contrôlés sont par définition disjointes.

Une fois les oracles introduits dans les composants du programme, il est possible d'appliquer la sémantique usuelle des programmes déterministes. En particulier, on sait détecter, pour les rejeter, les constructions qui produisent des systèmes non déterministes ou non réactifs. Pour un programme correct, on obtient une machine de Mealy booléenne déterministe, dans laquelle apparaissent encore les entrées oracles. Il suffit alors de les "effacer", pour ré-obtenir un objet non déterministe. Cette transformation en trois étapes peut constituer la sémantique d'un langage acceptant le non déterminisme dans les composants élémentaires.

Les exigences naturelles associées à ce schéma sont les suivantes :

– La sémantique en trois étapes ainsi définie ne devrait pas dépendre de la manière d'introduire les oracles. Autrement dit, quelle que soit la manière choisie pour introduire les

oracles lors de la phase 2, on devrait obtenir en bout de chaîne des machines comparables (égales, isomorphes, ... ?)

– Il devrait exister, sur les machines non déterministes, une relation d'équivalence constituant une congruence pour les opérateurs du langage.

4.2.2 Les machines contrôlées condition de séparation et résultat

Pour formaliser cette sémantique en trois étapes, on définit la notion de machine de Mealy booléenne *contrôlée*, dans laquelle les transitions portent des conditions de la forme $b_i \wedge b_o$. b_i représente la condition d'origine associée à la transition, et porte sur les véritables signaux d'entrée du système à décrire. b_o est une formule booléenne construite à l'aide de signaux *oracles*, introduite pour déterminiser la machine.

On décrit comment transformer tout programme ARGOS en programme dont les composants de base sont des machines contrôlées. Il faut déterminiser séparément les composants de base, en respectant quelques contraintes sur les ensembles de signaux oracles utilisés, comme cela a partiellement illustré par les exemples (des ensembles disjoints pour des composants parallèles, etc). La *condition de séparation* est une condition suffisante sur les formules booléennes introduites lors de la déterminisation d'un automate, qui garantit en particulier que la machine contrôlée obtenue est toujours réactive.

On prouve également que toutes les méthodes d'introduction des oracles qui respectent la condition de séparation garantissent des automates isomorphes en fin de chaîne (déterminisation – sémantique usuelle – effacement des oracles). Enfin, sous cette condition de séparation, on prouve que la bisimulation est une congruence pour les opérations d'ARGOS.

4.3 Recherche d'une autre relation d'équivalence

On a donc montré que la bisimulation est une congruence pour cette nouvelle sémantique en trois étapes. Ce résultat n'est qu'à moitié satisfaisant. Techniquement, exhiber une congruence est un minimum, pour justifier la sémantique. Il est heureux que la bisimulation convienne: c'est une relation bien connue, largement utilisée en association avec les algèbres de processus, et la comparaison (inclusion dans un sens ou dans l'autre) avec d'autres relations a été étudiée intensivement.

D'un point de vue plus philosophique, la seule relation de comparaison des systèmes non déterministes qui nous donnerait réellement satisfaction est définie, de manière tout à fait informelle, par: *un objet non déterministe représente un ensemble de comportements déterministes, et deux objets doivent être considérés comme équivalents s'ils représentent le même ensemble de comportements déterministes.* Puisque c'est l'interprétation du non déterminisme que nous avions en tête, on devrait pouvoir espérer que tout, dans la formalisation, est fait pour garantir que cette relation est une congruence. Toutefois nous avons rencontré dans la formalisation de cette sémantique des difficultés techniques inattendues, et il serait bon de pouvoir définir formellement cette nouvelle relation d'équivalence afin de prouver qu'elle constitue bien une congruence. Il n'est pas complètement exclu que cette conjecture soit fausse, d'ailleurs.

Cette interprétation du non déterminisme semble fort naturelle dans toute une communauté travaillant sur les spécifications non déterministes de systèmes parallèles, ainsi que l'atteste Manfred Broy. De même, un programme Prolog est dit non déterministe lorsqu'il peut produire

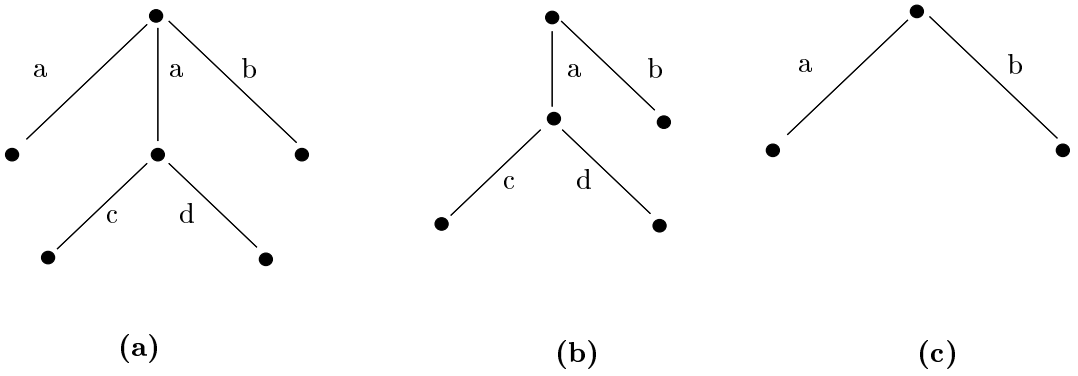


FIG. 4.5 – *Objets non déterministes et ensembles de comportements déterministes associés, dans le cas fini*

un ensemble de solutions (fourni en extension, ou décrit symboliquement par un ensemble de contraintes). En revanche, tous les spécialistes de la sphère "algèbres de processus" consultés jusque là avouent ne pas avoir rencontré de définition formelle d'une équivalence basée sur cette idée.

4.3.1 Définition d'une relation d'équivalence

Nous pouvons tenter de préciser quelque peu la définition informelle ci-dessus, avant de partir à la recherche d'exemples et de contre-exemples pour comparer notre relation d'équivalence avec des relations connues.

On observe facilement sur quelques exemples ce que peut être *l'ensemble de comportements déterministes décrit par un objet non déterministe*. Ce n'est pas un ensemble de comportements linéaires, ou traces — auquel cas l'équivalence cherchée serait simplement l'équivalence de trace — puisque les comportements d'entrée/sortie déterministes vis-à-vis des entrées peuvent présenter des branchements sur ces entrées. La figure 4.5 donne un exemple fini. **(a)** est un objet non déterministe qui représente l'ensemble { **(b)**, **(c)** }.

La situation se complique lorsque l'objet non déterministe comporte des boucles. En effet, l'ensemble de comportements déterministes associé est alors, dans le cas général, un ensemble *infini* de comportements déterministes à nombre *infini* d'états. Il suffit pour cela de placer un choix non déterministe dans une boucle. La situation semble maîtrisable puisqu'on dispose, par définition, d'une représentation concise et même finie de cet ensemble, mais il paraît pour l'instant difficile de définir l'équivalence recherchée directement.

Notons ND^1 et ND^2 des automates non déterministes et $E^1 = \{d_0^1, d_1^1, ...\}$, $E^2 = \{d_0^2, d_1^2, ...\}$ les ensembles potentiellement infinis — mais certainement dénombrables — des comportements déterministes associés. On dira que ND^1 et ND^2 sont équivalents par la relation cherchée notée $\equiv$ si et seulement si les ensembles E^1 et E^2 sont égaux modulo une équivalence de comportements déterministes, par exemple l'équivalence de traces notée $\sim$. Ainsi :

$$(ND^1 \equiv ND^2) \iff \begin{cases} \forall d_k^1 \in E^1, \exists d_j^2 \in E^2, d_k^1 \sim d_j^2 \\ \forall d_k^2 \in E^2, \exists d_j^1 \in E^1, d_k^2 \sim d_j^1 \end{cases}$$

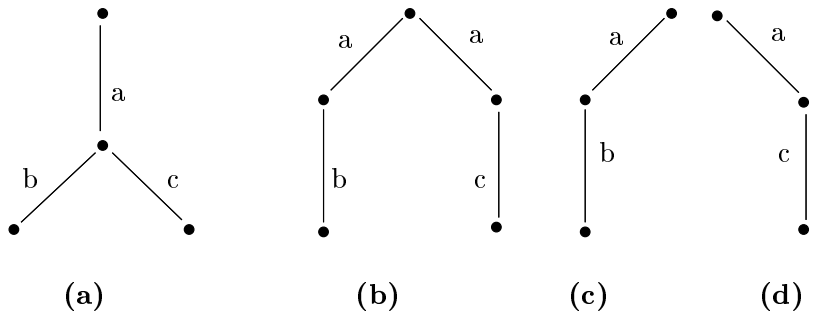


FIG. 4.6 – *Exemple prouvant que la relation recherchée n'est pas l'équivalence de trace*

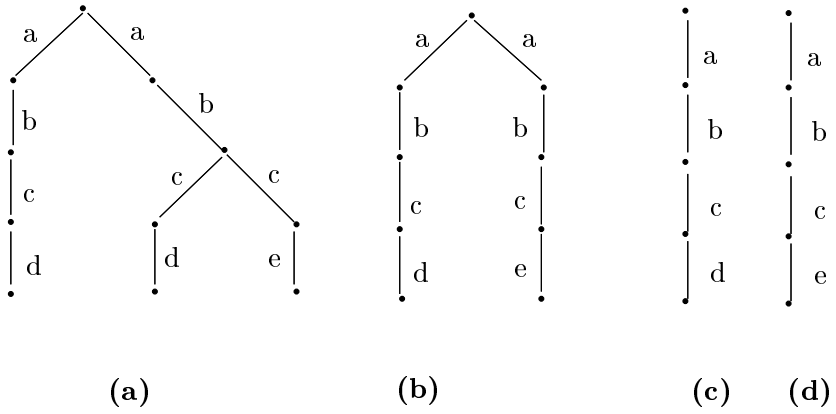


FIG. 4.7 – *Exemple prouvant que la relation recherchée n'est pas la bisimulation*

4.3.2 Essai de comparaison avec des relations d'équivalence connues

Exemple 4.5 : Ce n'est pas l'équivalence de trace

Les deux systèmes de transitions **(a)** et **(b)** de la figure 4.6 sont équivalents par l'équivalence de trace. Le premier, étant déterministe, représente un ensemble de comportements déterministes réduit à lui-même. Le second représente l'ensemble de comportements **(c)** et **(d)**.

□

Exemple 4.6 : Ce n'est pas la bisimulation

Les deux systèmes de transitions non déterministes **(a)** et **(b)** de la figure 4.7 ne sont pas bisimilaires, et pourtant ils représentent tous deux le même ensemble de comportements déterministes, à savoir les comportements **(c)** et **(d)** de la figure.

□

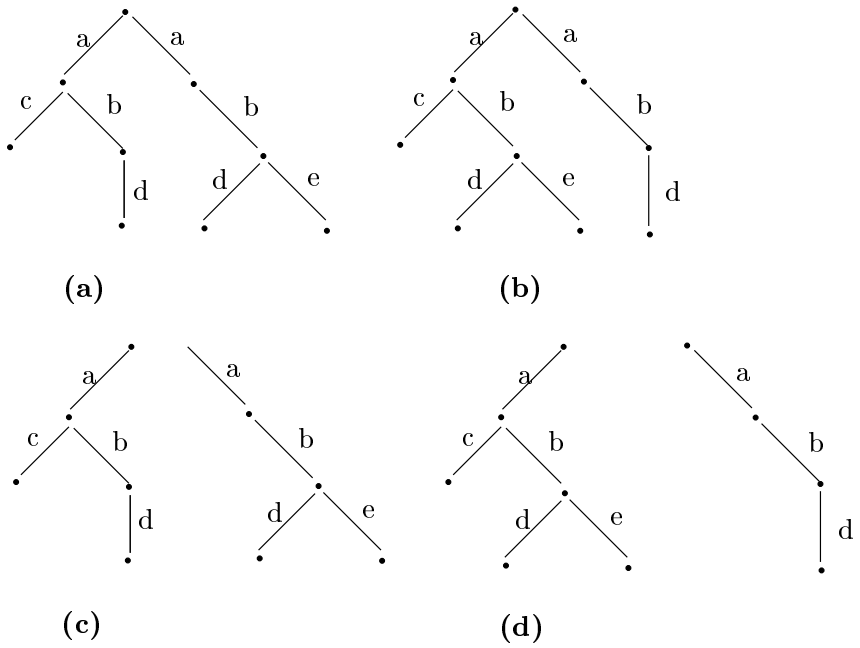


FIG. 4.8 – *Exemple prouvant que la relation recherchée n'est pas l'équivalence d'acceptation*

Exemple 4.7 : Ce n'est pas l'équivalence d'acceptation

Les automates **(a)** et **(b)** sont équivalents par l'équivalence d'acceptation, mais ils ne représentent pas les mêmes ensembles de comportements déterministes. En effet, **(a)** représente l'ensemble **(c)**, et **(b)** représente l'ensemble **(d)**. □

4.4 Application à d'autres langages synchrones, autres applications de la technique des oracles

L'introduction annonçait que les résultats de ce travail permettent de ne plus considérer les langages synchrones comme étant voués au déterminisme. L'exposé est centré sur ARGOS, où le non déterminisme intrinsèquement voulu par le programmeur apparaît très naturellement dans les automates de base du programme.

Appliquer la démarche à un autre langage synchrone exige que l'on décide préalablement de l'introduction d'un brin de non déterminisme dans les programmes. Selon les structures proposées par le langage, ce n'est pas forcément évident.

Pour LUSTRE, N. Halbwachs propose l'introduction de variables locales non définies par des équations. En effet, jusque là, la modélisation du non déterminisme est réalisée en introduisant des entrées oracles au plus haut niveau du programme, qui doivent être passées en paramètres de tous les noeuds internes, jusqu'au noeud où elles jouent effectivement le rôle d'oracles. Disposer de variables locales non définies par des équations permet de ne mentionner ces oracles qu'à l'endroit où ils sont utilisés. Il reste à la charge du compilateur de les traiter comme des entrées.

Enfin la technique des oracles peut être utilisée dès que l'on doit simuler ou coder le non déterminisme dans un contexte prévu à l'origine pour traiter le cas déterministe. Considérons par exemple le codage des comportements de systèmes réactifs par des BDD. Il est connu que les *fonctions* sont représentées et manipulées de manière bien plus efficace que les *relations*. Or, si les transitions d'un système sont données par des fonctions et non par des relations, c'est que le système est déterministe. Pour coder efficacement des systèmes non déterministes (par exemple la relation de transition inverse d'un système déterministe, souvent utilisée dans les algorithmes de vérification dits "*en arrière*") il suffit d'introduire des oracles, comme proposé par Pascal Raymond.

Chapitre 5

Sémantique d'Argos en équations booléennes

Sources : *Le codage d'*ARGOS *en équations booléennes, tel que présenté ici, est publié dans [MH96a]. Toutefois les idées suivies avaient déjà été utilisées pour le codage d'*ARGOS *en* LUSTRE *présenté dans la thèse de Muriel Jourdan [Jou94], ou dans [JLMR93, JLMR94], à propos de programmation multi-langages. D'autre part l'analyse de déterminisme et réactivité présentée dans [HM95] a été expérimentée grâce à un codage d'*ARGOS *en équations booléennes au format d'entrée de l'outil* BAC *(Boolean Automaton Checker) développé par N. Halbwachs.*

La sémantique des constructions d'ARGOS présentée au chapitre 2 ne peut guère servir de technique de compilation. La taille de l'automate plat obtenu par composition des automates de base d'un programme est naturellement une fonction exponentielle de la taille du programme, du fait des compositions parallèles interprétées comme des produits cartésiens. D'autre part, même si la taille de l'objet final n'est pas prohibitive, la taille des objets intermédiaires peut l'être.

Enfin, pour appliquer une opération d'encapsulation, on doit résoudre un système d'équations booléennes dont les variables sont les signaux locaux, et cela pour chaque état, et pour chaque configuration possible des signaux d'entrée !

Le tout premier compilateur ARGOS résolvait ces systèmes par essai exhaustif de toutes les configurations de valeurs de variables (cela ne relevait toutefois pas d'un choix délibéré, mais d'une compréhension encore incomplète des relations entre diffusion synchrone et systèmes d'équations). Dans le second compilateur développé par Muriel Jourdan, la résolution des systèmes était réalisée grâce à un codage en BDD, la production de l'ensemble des états étant toujours réalisée de manière explicite par un parcours en avant (la technique est décrite dans [MV92b]).

L'étape suivante est le codage complet des programmes ARGOS en systèmes d'équations, pour permettre un calcul symbolique. Cela demande une refonte complète de la technique de compilation, mais c'est une évolution indispensable. Ce codage en équations a été utilisé tout d'abord pour réaliser une détection symbolique des cas de non déterminisme ou non réactivité, dans [HM95]. La technique utilisée ne conduit pas directement à la génération de code. Il s'agissait plutôt d'expérimenter les algorithmes BDD, et de comparer la notion de correction obtenue avec les notions dites "*de causalité*" définies par G. Berry pour ESTEREL, ou connues dans le domaine des circuits séquentiels (voir [Mal93] par exemple).

Le format DC [CS95], qui constitue l'un des formats communs de la plate-forme synchrone, permet de décrire les systèmes réactifs par des ensembles d'équations, dans le style de LUSTRE.

Toutefois, LUSTRE propose une notion générale d'*horloge*, alors que DC se contente d'une notion de *condition d'activation*. L'exécution d'un programme DC peut être vue comme l'évaluation répétée d'un ensemble d'affectations parallèles (définies par les équations). Il est possible, grâce aux conditions d'activation, de spécifier qu'il n'est pas nécessaire d'évaluer une certaine équation à tous les cycles. L'opération de composition hiérarchique d'ARGOS constitue précisément un moyen de spécifier que certaines parties du programme peuvent ne pas toujours être *actives*, au cours de l'exécution. Il est donc particulièrement intéressant de traduire ARGOS en DC, en essayant d'utiliser au mieux ces conditions d'activation.

5.1 Argos vers DC

5.1.1 un sous-ensemble du format DC

EQU(i, b, a) définit un flot i, comme étant égal au résultat de l'évaluation de l'expression booléenne b, mis à jour lorsque a est vrai :

$$i_0 = \begin{cases} b_0 \text{ si } a_0 \\ v \text{ sinon} \end{cases}, \qquad i_n = \begin{cases} b_n \text{ si } a_n \\ i_{n-1} \text{ sinon} \end{cases} \quad \text{Quand } n > 0$$

La valeur initiale v utilisée dans la définition devrait apparaître en paramètre d'une définition. Toutefois cette valeur ne sert que lorsque la condition d'activation est fausse à l'instant initial. Dans la traduction d'ARGOS en DC, un flot défini par un EQU, et dont la condition d'activation est fausse à l'instant initial, n'est en fait utilisé nulle-part à cet instant. Les valeurs initiales des EQU sont donc non pertinentes.

MEM(i, b, a, v) définit un flot i comme la mémorisation du flot calculé par l'expression booléenne b, mise à jour lorsque a est vrai. La valeur initiale de i est donnée par v.

$$i_0 = v, \qquad i_{n+1} = \begin{cases} b_n \text{ si } a_n \\ i_n \text{ sinon} \end{cases}$$

La figure 5.1 illustre les deux types de définitions. Les définitions de type MEM reproduisent leur entrée avec un décalage d'un instant. Les définitions de type EQU reproduisent leur entrée instantanément.

5.1.2 Utilisation des deux types de définitions : Codage des machines de Mealy

Le codage des machines de Mealy en définitions de type MEM ou EQU est réalisé en décidant d'utiliser une variable de flot booléen par signal, et une variable par état. On pourrait bien sûr optimiser le nombre de variables d'état par un codage en log, mais c'est alors le codage des transitions qui devient complexe. D'autre part les machines de Mealy qui interviennent comme composants de base des programmes ARGOS n'ont en général que peu d'états. Pour une dizaine d'états, il semble peu intéressant de compliquer la fonction de transition par un codage en log des états, qui ne représente qu'une petite économie de variables d'états. Une réutilisation partielle des variables d'état, guidée par la structure du programme ARGOS, est un objectif bien plus intéressant (voir plus loin le paragraphe 5.1.6).

La figure 5.2 donne un exemple de machine (a), les chronogrammes correspondants (b) et la correspondance entre instants du chronogramme et transitions exécutées (c). Pour cet exemple, les définitions sont :

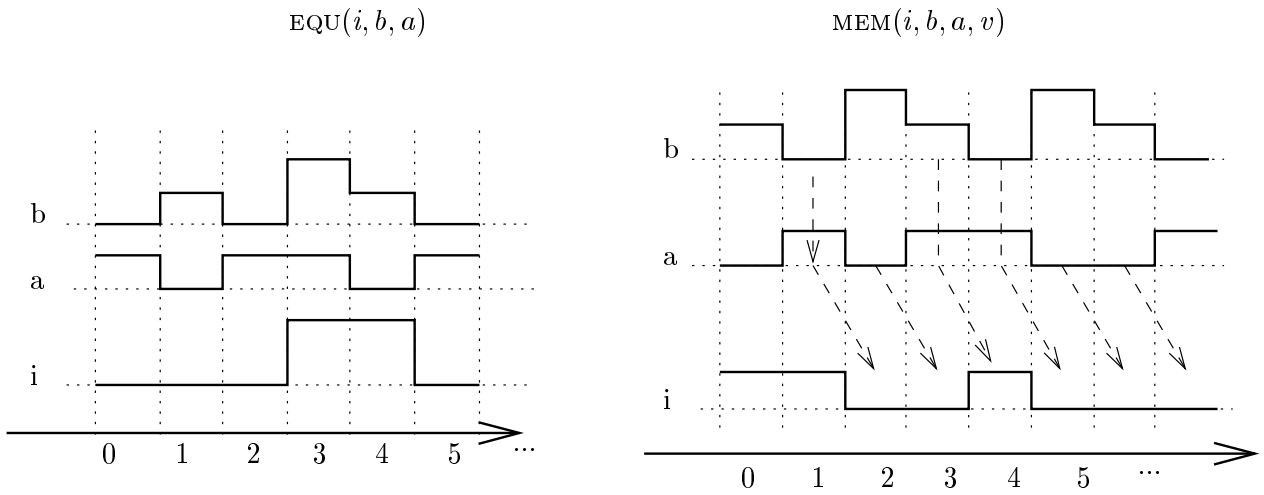


FIG. 5.1 – *Exemples de définitions DC et chronogrammes correspondants*

MEM$(A, A \wedge \neg a \vee B \wedge a,$ TRUE, TRUE$)$	EQU$(b, A \wedge a,$ TRUE$)$
MEM$(B, B \wedge \neg a \vee A \wedge a,$ TRUE, FALSE$)$	EQU$(c, A \wedge \neg a,$ TRUE$)$

Si une machine de Mealy est considérée comme constituant un programme, les conditions d'activation doivent être toujours vraies : considérons une machine (S, s_0, I, O, T). Pour chaque état $s \in S$ on définit un flot local s par : MEM$(s, rp,$ TRUE, initv$)$ où

$$rp = \left[(s \wedge \neg \bigvee_{(s,b,-,-) \in T} b) \vee \bigvee_{(s',b,-,s) \in T} (s' \wedge b) \right]$$

De plus, si l'état est initial (c'est-à-dire $s = s_0$), initv = TRUE, sinon initv = FALSE.
Pour chaque signal de sortie $o \in O$, on définit un flot local o par : EQU$(o, rp,$ TRUE$)$ où

$$rp = \left[\bigvee_{(s,b,O \ni o,-) \in T} (s \wedge b) \right]$$

5.1.3 Codage des programmes structurés

Le codage des programmes structurés est guidé par la syntaxe : l'ensemble de définitions d'un programme $P_1 \| P_2$ est construit à partir des ensembles de définitions obtenus pour P_1 et P_2 indépendamment. Le codage complet est fourni dans [MH96a]. J'expose ci-dessous les principales difficultés et les idées de solution.

Composition parallèle et émetteurs multiples d'un signal. Lorsque deux composants parallèles émettent le même signal o, on obtient deux ensembles de définitions non disjoints, au sens où leur union contient deux définitions du même flot. Dans le format DC, comme en LUSTRE, chaque flot est défini par une seule équation. L'interprétation correcte de la sémantique

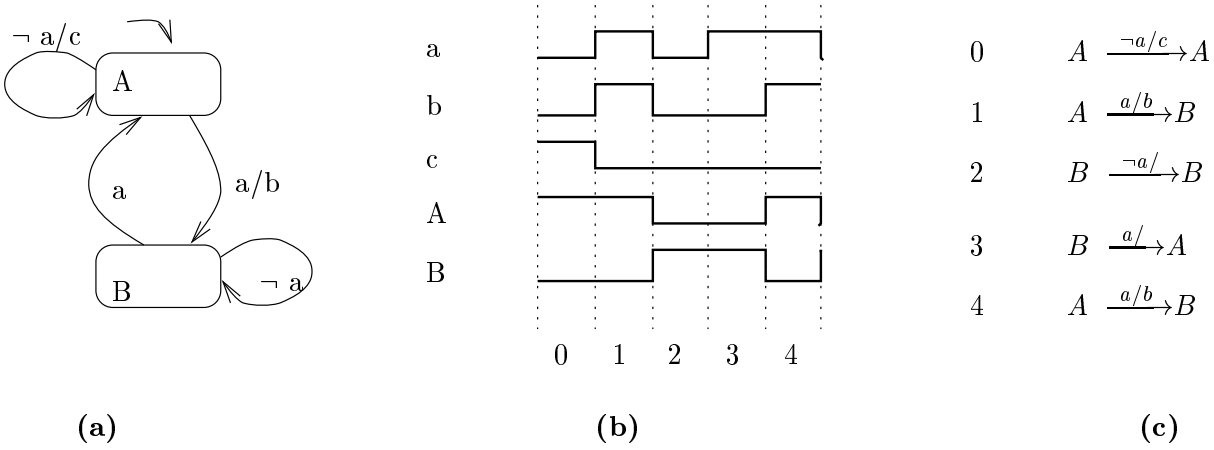


FIG. 5.2 – *Codage des machines de Mealy, exemple*

d'ARGOS consiste à fusionner les deux définitions en prenant la disjonction de leurs parties droites. (En ESTEREL, où les signaux peuvent porter des valeurs de types quelconques, une émission multiple s'accompagne de la combinaison des valeurs associées, par une opération commutative et associative.)

Codage du raffinement, les difficultés. La difficulté majeure provient du codage du raffinement. D'une part, c'est une bonne idée d'utiliser les opérations de raffinement comme des indications permettant de restreindre les conditions d'activation des différents flots calculés : typiquement, il est inutile de calculer les flots d'états d'un processus qui raffine un état non actif. D'autre part, une application brutale de ce principe conduit à une traduction fausse. D'une certaine manière, les flots définissant les signaux *doivent* être calculés à tout instant. Enfin il faut garantir la réinitialisation des processus raffinants ; il ne suffit pas d'inhiber le calcul des variables d'états d'un processus non actif, il faut également garantir que le processus redémarre dans son état initial, s'il est tué puis relancé.

Tous ces points sont garantis par un codage *sous contexte* des expressions de programmes ARGOS. Chaque sous-arbre est codé en tenant compte de deux paramètres hérités notés KILL et ALIVE, qui signifient respectivement : *le processus représenté par ce sous-arbre est tué durant la réaction courante* ; *le processus participe à la réaction courante.*

Les paramètres sont définis a priori pour le programme principal, c'est-à-dire la racine de l'arbre abstrait : KILL = FALSE, puisque le programme principal n'est jamais tué, et ALIVE = TRUE, puisqu'il participe à toutes les réactions. On définit comment ces paramètres sont passés d'un noeud à ses fils, et comment les feuilles (les machines de Mealy qui contrôlent les raffinements) sont codés en fonction de ces paramètres hérités.

Codage des machines de Mealy sous contexte. L'idée de base du codage est de maintenir l'invariant : *toute machine appartenant à un processus non actif est mise dans son état initial.* C'était le principe de base des premières sémantiques opérationnelles d'ARGOS, qui garantissait la production de systèmes de transitions minimaux selon un certain critère [Mar90]. Pour garantir cet invariant, les machines appartenant à un processus sont remises à leur état initial lorsque ce

processus est tué.

On obtient donc, pour un état s, la définition MEM(s, rp, Alive, initv), dans laquelle :

$$rp = \left[(s \wedge \neg \bigvee_{(s,b,-,-)\in T} b) \vee \bigvee_{(s',b,-,s)\in T} (s' \wedge b) \right] \begin{cases} \vee \text{ Kill} & \text{si } s \text{ est initial} \\ \wedge \neg \text{ Kill} & \text{sinon} \end{cases}$$

Comme auparavant, initv = TRUE si s est initial, sinon initv = FALSE.

Pour un signal o, la seule modification concerne la condition d'activation : EQU(o, rp, Alive).

Codage du raffinement. Le codage faux contre lequel il faut se prémunir est le suivant : si l'on ne calcule pas les sorties d'un processus P lorsque P est inactif, on obtient un comportement erroné des signaux émis par P. En effet, si un signal o doit être émis lors de la réaction qui tue P, la valeur de son flot associé est calculée, mise à vrai, et restera vraie pendant que P est inactif.

La solution choisie consiste à associer à chaque signal deux flots, calculés à des niveaux différents : P calcule une version partielle du signal o, notée o', et ceci peut être fait avec comme condition d'activation une expression définissant les instants d'activité de P ; dans le contexte englobant (qui définit les instants d'activation de P), on calcule o par EQU($o, a \wedge o', x$), où x est la condition d'activation du contexte.

5.1.4 Exemple

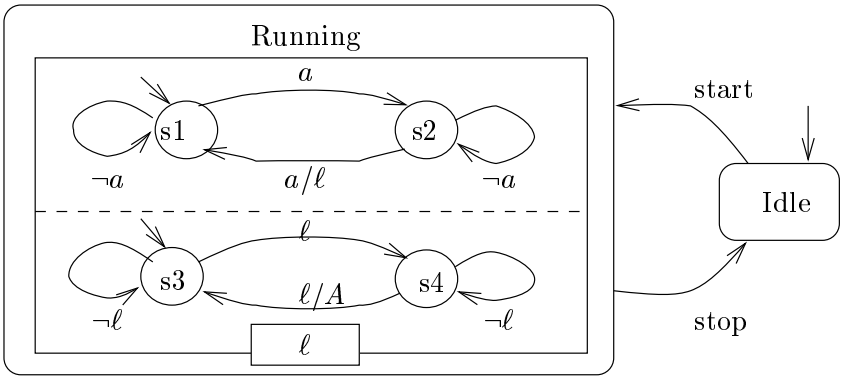


FIG. 5.3 – *Un programme simple à coder en* DC

Les définitions correspondantes :

$$\text{MEM}(\mathsf{Running}, \mathsf{Running} \wedge \neg\mathsf{Stop} \vee \mathsf{Idle} \wedge \mathsf{Start}, \text{TRUE}, \text{FALSE})$$
$$\text{MEM}(\mathsf{Idle}, \mathsf{Idle} \wedge \neg\mathsf{Start} \vee \mathsf{Running} \wedge \mathsf{Stop}, \text{TRUE}, \text{TRUE})$$

$$\text{MEM}(s_1, s_1 \wedge \neg a \vee s_2 \wedge a \vee \mathsf{Stop}, \mathsf{Running}, \text{TRUE})$$
$$\text{MEM}(s_2, s_2 \wedge \neg a \vee s_1 \wedge a, \mathsf{Running}, \text{FALSE})$$
$$\text{MEM}(s_3, s_3 \wedge \neg \ell' \vee s_4 \wedge \ell' \vee \mathsf{Stop}, \mathsf{Running}, \text{TRUE})$$
$$\text{MEM}(s_4, s_4 \wedge \neg \ell' \vee s_3 \wedge \ell', \mathsf{Running}, \text{FALSE})$$

$$\text{EQU}(\ell', s_2 \wedge a, \mathsf{Running})$$
$$\text{EQU}(A', s_4 \wedge \ell', \mathsf{Running})$$

$$\text{EQU}(\ell, \mathsf{Running} \wedge \ell', \text{TRUE})$$
$$\text{EQU}(A, \mathsf{Running} \wedge A', \text{TRUE})$$

5.1.5 Propriété de cohérence

La sémantique opérationnelle décrite au chapitre 2 donne une machine de Mealy (S, s_0, I_a, O_a, T). La nouvelle sémantique définie ici donne un programme DC sous la forme (I_d, O_d, L, D) (I_d et O_d sont les ensembles de flots d'entrée et de sortie, L est l'ensemble des flots locaux, et D est l'ensemble des définitions). Il faut trouver un dénominateur commun à ces deux objets afin de les comparer.

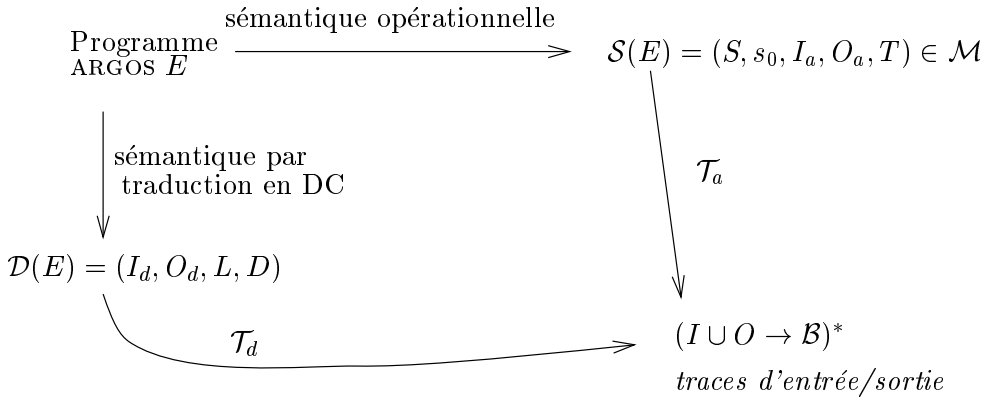


FIG. 5.4 – *Propriété de cohérence des deux sémantiques*

On les considère tous deux comme des générateurs d'ensembles de traces d'entrés/sortie, c'est-à-dire des suites de fonctions de $I \cup O$ vers les booléens. Il est facile de se convaincre, en effet, que les deux objets sémantiques s'accordent sur la notion d'entrée/sortie, et donc $I = I_a = I_d$ et, de même, $O = O_a = O_d$. Il faut donc prouver $\forall E \in \mathcal{E} \quad \mathcal{T}_a(\mathcal{S}(E)) = \mathcal{T}_d(\mathcal{D}(E))$.

On prouve plus facilement une propriété plus forte, selon laquelle ce sont en fait les traces d'entrées/sorties/états qui coïncident. Il suffit d'établir la correspondance entre les états S de la machine de Mealy et les flots définis par des MEM dans le programme DC (en tenant compte des flots dont la valeur n'est pas pertinente à un instant donné parce qu'ils appartiennent à des

processus non actifs). On retrouve ce type de raisonnement pour prouver l'équivalence de deux sémantiques dans l'article sur l'extension temporelle d'ARGOS [JMO93].

5.1.6 Complexité de la traduction

La taille du programme DC obtenu est linéaire par rapport à la taille du programme ARGOS de départ.

Retour sur le codage des états Nous avons décrit une technique de codage qui alloue un ensemble de variables pour chaque machine de Mealy composant de base. La structure de raffinement conduit naturellement à envisager une réutilisation partielle des variables d'état, guidée par la structure du programme. D. Drusinsky [DY90] a proposé un algorithme de synthèse de circuit à partir d'un Statechart, en identifiant des ensembles les plus larges possible d'états élémentaires dits *exclusifs*, pour lesquels on peut choisir un codage en log. Cette technique n'a d'intérêt que si les ensembles considérés ne sont pas limités aux ensembles d'états des machines de Mealy de base, et pour les élargir il faut tenir compte de la structure du programme.

Faut-il pour autant compliquer l'algorithme de codage — au risque d'y introduire des erreurs, ou du moins de le rendre difficilement prouvable —, ou s'en remettre à l'application a posteriori d'un algorithme d'élimination de variables du style de ceux utilisés dans les outils de conception de circuits ? Cette dernière approche est celle d'ESTEREL. Des algorithmes spécifiques ont toutefois été étudiés, capables de réaliser une réduction conséquente pour la classe de circuits obtenus comme traduction de programme ESTEREL.

Dans le cas d'ARGOS, des expériences restent à faire pour déterminer l'efficacité des algorithmes existants sur les circuits produits. Il est possible que l'idée de placer les sous-programmes inactifs dans leur état initial ait pour effet de dissimuler l'exclusivité de certaines variables d'état, et donc d'interdire une réduction sensible. L'idéal serait un algorithme d'élimination de variables dans un circuit booléen, capable de tenir compte d'un ensemble de contraintes d'exclusivité fourni par le compilateur ARGOS.

5.2 Analyse de déterminisme et réactivité sur la forme équationnelle

Dans [MH96a], nous avons délibérément omis l'aspect de détection des programmes incorrects (non déterministes ou non réactifs). Pour que la traduction en DC proposée ici fournisse une nouvelle méthode de compilation d'ARGOS, il faut savoir intégrer la détection des programmes incorrects.

Dans [HM95], nous montrions comment réaliser l'analyse de déterminisme et réactivité sur la forme équationnelle des programmes. Reste donc à savoir où intégrer cette analyse, dans la chaîne de compilation (voir aussi le chapitre 13, pour un schéma global de la chaîne de compilation).

Je présente ici rapidement les résultats de [HM95], et la manière d'intégrer l'analyse qui y est décrite dans la compilation d'ARGOS.

5.2.1 Cadre de l'étude

On considère des *automates booléens*, dont les transitions sont définies de manière symbolique par des ensembles d'équations. Ces équations peuvent faire intervenir des variables dites *locales*, et présenter des boucles de définition.

Definition 5.1 : automate booléen
Un automate booléen à n variables d'états, m variables d'entrée, p variables locales et q variables de sortie est un n-uplet $A = (init, \Sigma, \Omega, \Lambda, \mathcal{C})$ où :

- $init \subseteq B^n$ est l'ensemble des états *initiaux* ($B = \{0, 1\}$)
- Σ est une fonction totale de B^{m+n+p} vers B^n (la fonction de transition), donnée par un vecteur $(\Sigma_k)_{k=1...n}$ de fonctions de B^{m+n+p} vers B. Si $s \in B^n$, $i \in B^m$, $\ell \in B^p$, $\Sigma(s, i, \ell)$ représente le vecteur $[\Sigma_k(s, i, \ell)]_{k=1...n}$.
- De même, Ω est une fonction totale de B^{m+n+p} vers B^q (la fonction de sortie), donnée par un vecteur $(\Omega_k)_{k=1...q}$ de fonctions de B^{m+n+p} dans B.
- Λ est une fonction totale de B^{m+n+p} dans B^p (la fonction de définition des variables locales), donnée par un vecteur $(\Lambda_k)_{k=1...p}$ de fonctions de B^{m+n+p} dans B.
- $\mathcal{C} \subseteq B^{m+n}$ donne les contraintes sur l'environnement (cf. ci-dessous).

□

Un tel automate peut être vu comme une machine $M = (S, I, O, T)$ pourvu des contraintes $\mathcal{C}$ sur l'environnement, avec :

$$S = B^n, \quad I = B^m, \quad O = B^q$$
$$(s, i/o, s') \in T \iff \exists \ell \in B^p \text{ tel que } \ell = \Lambda(s, i, \ell) \text{ et } s' = \Sigma(s, i, \ell) \text{ et } o = \Omega(s, i, \ell).$$

On appelle *configuration* du système un couple (état, entrée). Une contrainte sur l'environnement est une formule booléenne des variables d'entrée et d'état du système, qui décrit les configurations dont l'environnement garantit l'inaccessibilité. Les contraintes les plus simples ne portent que sur les variables d'entrée, et peuvent servir par exemple à exprimer que deux signaux ne surviennent jamais simultanément. Une contrainte plus compliquée, nécessitant la prise en compte des variables d'état, peut exprimer que deux signaux d'entrée alternent. Dans le domaine de la conception de circuits, on parle de "*don't care sets*".
La définition d'un automate booléen comprend trois systèmes d'équations : le système $s'_k = \Sigma_k(s, i, \ell), k = 1...n$ définit les nouvelles valeurs des variables d'états ; le système $o_k = \Omega_k(s, i, \ell), k = 1...q$ définit les sorties courantes ; le système $\ell_k = \Lambda_k(s, i, \ell), k = 1...p$ définit les valeurs courantes des variables locales.
Les variables de sortie et les variables d'états sont définies par des fonctions *totales* de l'état courant, de la valeur courante des entrées, et de la valeur courante des variables locales. Seules les variables locales peuvent être définies par des équations cycliques.

5.2.2 Diverses notions de cohérence

Considérons la fonction *loc*, de B^{m+n} dans 2^{B^p} qui associe à chaque configuration (s, i) l'ensemble des solutions du système d'équations $\ell = \Lambda(s, i, \ell)$. Un automate booléen sera dit *fortement cohérent* si et seulement si, pour chaque configuration accessible (s, i), $\mid loc(s, i) \mid = 1$. Il sera dit *faiblement cohérent* si et seulement si, pour chaque configuration accessible (s, i), $\mid \{\Omega(s, i, \ell), \ell \in loc(s, i)\} \mid = 1$ et $\mid \{\Sigma(s, i, \ell), \ell \in loc(s, i)\} \mid = 1$.

Autrement dit, dans un automate fortement cohérent, la valeur des variables locales est déterminée de manière unique, dans chaque configuration accessible. Dans un automate faiblement

cohérent, certaines variables locales peuvent ne pas être déterminées, si cela n'influence ni les sorties, ni l'état suivant.

5.2.3 Résultat

Nous décrivons dans [HM95] une méthode de vérification de la cohérence forte, qui s'applique à un système d'équations $\ell = \Lambda(s, i, \ell)$ éventuellement cyclique et fournit deux résultats :

- Une condition $E(s, i)$ vraie exactement quand $\mid loc(s, i) \mid = 1$
- Un nouveau système d'équations $\ell = \Lambda'(s, i)$, non cyclique, et tel que $E(s, i) \Longrightarrow loc(s, i) = \{\Lambda'(s, i)\}$

Ce résultat est l'application d'un théorème d'algèbre de Boole très utilisé dans la conception de circuits pour éliminer des variables dans les fonctions booléenees. L'idée est qu'on peut éliminer une variable si l'on sait prouver qu'elle est entièrement déterminée comme fonction des autres variables. On essaie donc d'éliminer successivement toutes les variables locales — sur lesquelles portent des équations cycliques.

Le théorème utilisé Le théorème s'énonce comme suit :

$$(F \implies (x = f)) \iff \neg(F_x \wedge F_{\overline{x}})$$

où F est une fonction de x et d'autres variables, et f une fonction des autres variables seulement. F_x et $F_{\overline{x}}$ proviennent de la décomposition de Shannon de F selon x :

$$F = (x \wedge F_x) \vee (\overline{x} \wedge F_{\overline{x}})$$

Pour exhiber une fonction f solution, on peut choisir toute fonction booléenne g comprise entre F_x et $\neg F_{\overline{x}}$, c'est-à-dire telle que $F_x \Rightarrow g \Rightarrow \neg F_{\overline{x}}$.

Méthode On code tout le système d'équations de l'automate booléen par une formule F dans laquelle apparaissent, en particulier, les variables locales $\{\ell_0, \ell_1, ..., \ell_k\}$. On essaie de prouver que, pour chacune d'entre elles, F détermine une solution unique.

Puisque $F = x \wedge F_x \vee \overline{x} \wedge F_{\overline{x}}$, $F_x \vee F_{\overline{x}}$ est une condition nécessaire et suffisante pour qu'il existe une solution pour la variable x dans le système F. Pour qu'elle soit de plus unique, d'après le théorème cité ci-dessus, il faut et il suffit que $\neg(F_x \wedge F_{\overline{x}})$.

Partant de $F^{(0)} = F$, on tente d'éliminer chaque variable locale l_i à son tour, et on calcule donc successivement, pour $i \in [1..k]$:

$$\begin{aligned} F^{(i)} &= F_{l_i}^{(i-1)} \vee F_{\overline{l_i}}^{(i-1)} \\ g^{(i)} &= F_{l_i}^{(i-1)} \\ E^{(i)} &= \neg(F_{l_i}^{(i-1)} \wedge F_{\overline{l_i}}^{(i-1)}) \end{aligned}$$

Ensuite on substitue la solution $g^{(j)}$ choisie pour la variable l_j à toute occurrence de l_j dans les formules $g^{(i)}$ et $E^{(i)}$ où $j > i$. Le résultat est formé du système d'équations $l_i = g^{(i)}, i = 1..k$, et de la formule $E = \bigwedge_{i=1}^{k} E^{(i)}$, qui ne porte plus que sur les variables d'état et les variables d'entrée. E est une condition nécessaire et suffisante pour que F détermine entièrement toutes les variables locales.

Note : ces algorithmes étant implantés en utilisant des BDD, on ne choisit pas $g^{(i)} = F_{l_i}^{(i-1)}$ mais plutôt $g^{(i)} = F_{l_i}^{(i-1)} \uparrow F^{(i)}$, où $\uparrow$ représente l'opérateur de restriction défini dans [BCM90]. Cette fonction est bien comprise entre F_{ℓ_i} et $\neg F_{\overline{\ell_i}}$, et l'opération de restriction garantit un BDD plus petit.

On trouvera dans [HM95] l'exposé de la méthode d'analyse pour la cohérence faible.

5.2.4 Intégration dans le processus de compilation d'ARGOS

Dans l'expérimentation conduite avec l'outil BAC, un programme ARGOS est traduit globalement en un automate booléen comme défini ci-dessus, et l'analyse de déterminisme et réactivité est réalisée globalement (aux conditions d'activation près, la traduction suit les idées de la traduction d'ARGOS en DC). Cela revient à considérer que tous les signaux déclarés locaux à un sous-arbre sont en fait "remontés" jusqu'à la racine de l'arbre abstrait. Cela suppose un renommage, puisqu'un même nom de signal peut être utilisé dans deux sous-arbres distincts. La question importante concerne la sémantique.

La figure 5.5 donne un exemple de programme dans lequel un cas de non-déterminisme détecté dans le sous-arbre d'une opération d'encapsulation est ensuite levé par le contexte. Si l'on applique strictement la sémantique décrite au paragraphe 2.3, page 35, le programme est déclaré globalement incorrect, puisque la valeur $\perp$ obtenue comme sémantique du composant qui déclare a, b locaux se propage jusqu'à la racine.

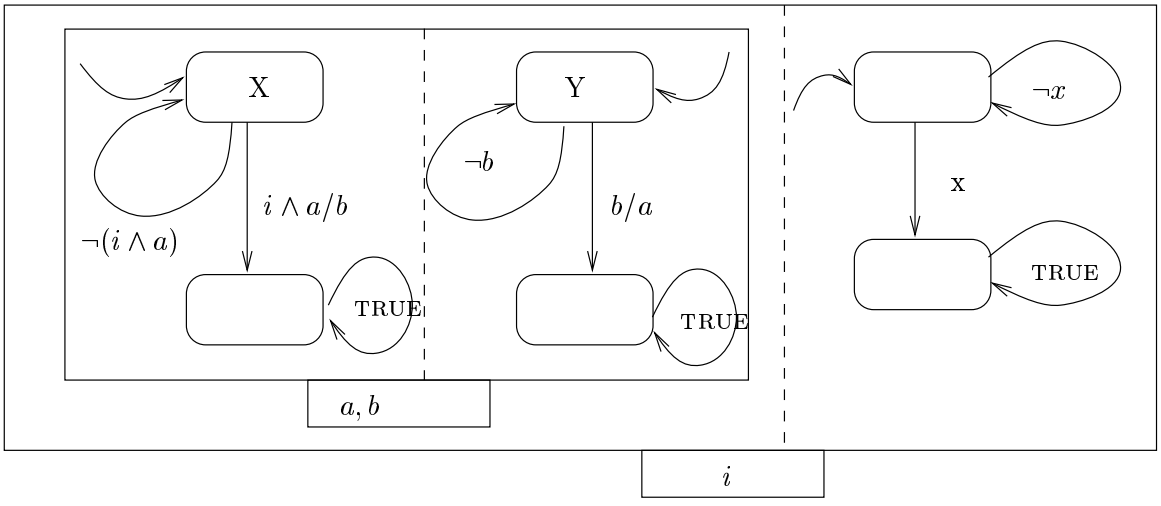


FIG. 5.5 – *Un exemple de non déterminisme éliminé par le contexte*

Si l'on applique les techniques décrites plus haut, l'analyse du sous-arbre incriminé ne produit pas la valeur $\perp$, mais une information plus riche : la condition sur les états et signaux d'entrée qui garantit que le programme est réactif et déterministe. Dans le cas de l'exemple, on obtiendra la condition $\neg(X \wedge Y \wedge i)$ — en supposant que chaque état est codé par une variable de même nom. On obtient également les équations qui définissent a et b en fonction des autres variables, dans les cas où cette condition est vérifiée, ici : $a = b =$ FALSE.

On peut ainsi, pour chaque sous-arbre d'opération d'encapsulation, produire une condition

nécessaire et suffisante de correction, et un ensemble d'équations définissant les signaux de sortie et les états en fonction des siganux d'entrée, dans lequel n'apparaissent plus de variables locales.

Les conditions sont propagées dans le contexte, où elles peuvent devenir trivialement vraies, comme c'est le cas dans l'exemple ci-dessus : le contexte fournit l'équation $i = \text{FALSE}$.

A la racine de l'arbre, on obtient une condition globale. Si elle est impliquée par les contraintes sur l'environnement (exclusivité de signaux, ...), le programme peut être considéré comme correct. Dans le cas contraire, la condition peut être vraie pour des raisons d'accessibilité des états qui y interviennent. Dans ce cas l'analyse exacte de correction demande un calcul des états accessibles. On peut aussi se contenter de ce résultat trop fort, et considérer qu'une erreur de déterminisme ou de causalité dissimulée dans une partie inaccessible du programme mérite d'être signalée... et corrigée.

On retrouve ici la discussion sur la détection des erreurs de type dans les portions de code mort, qui se pose pour les langages séquentiels. A ceci près qu'une portion de code mort détectable dans un langage séquentiel peut en général être supprimée syntaxiquement, alors que ce n'est pas le cas pour ARGOS (pour un langage à structure parallèle en général) : on peut avoir détecté l'inaccessibilité d'une configuration d'états des divers composants parallèles, et ne pouvoir supprimer les états incriminés, qui participent à d'autres configurations parfaitement accessibles, elles.

Note : une comparaison de cette définition de correction des programmes avec la définition de *causalité* introduite pour ESTEREL est esquissée au chapitre 9.

Chapitre 6

Eléments de définition du langage Argos

Sources : *Non publié. En partie implémenté.*

Le chapitre 2 présente la base d'ARGOS, sous forme d'un ensemble d'opérations sur les machines de mealy booléennes. La définition de ce qui pourrait commencer à s'appeler un *langage*, à partir de cette base formelle, exige que l'on s'intéresse à des aspects aussi divers que le choix d'une syntaxe concrète (qu'elle soit graphique ou textuelle) ; l'introduction d'un mécanisme de structuration des textes de programmes ; le support pour une répartition des programmes sur plusieurs fichiers ; la définition de macro-notations pour les constructions d'usage courant ; un support pour les aspects de remontée au source lors du processus de compilation ; un support à toutes les extensions envisagées (présentées dans les chapitres suivants). Nous donnons ici une proposition de syntaxe concrète qui tient compte de la plupart de ces aspects.

6.1 Structuration des programmes et restriction de l'environnement

6.1.1 Structuration

Les programmes ARGOS seront structurés en *processus* comme les programmes LUSTRE en *noeuds*. L'idée est la même, le mot *processus* étant choisi simplement pour refléter le caractère plus impératif du langage.

Un processus a un *nom*, qui permet de l'appeler dans un autre contexte ; une interface, c'est-à-dire deux listes de paramètres formels (d'entrée et de sortie), qui désignent des signaux booléens ; une liste de signaux locaux ; et un *corps* : c'est une construction à l'aide des opérateurs ARGOS définis au chapitre 2, dans laquelle les étiquettes de transitions d'automates sont construites en utilisant les signaux déclarés dans l'interface ou locaux au processus.

6.1.2 Restriction sur l'environnement d'un processus

Nous avons mentionné au chapitre 5 la possibilité d'associer à un noeud DC une spécification des configurations dont l'environnement garantit l'inaccessibilité. Une *configuration* du système est un couple (état, entrée). En DC, une telle spécification est une simple formule sur les variables locales définies par des mémorisations — les états — et sur les variables d'entrée. En ESTEREL

on trouve deux types de contraintes instantanées sur les signaux d'entrées d'un module: des exclusions n-aires (notées $\sharp$) et des implications. En LUSTRE le mécanisme des assertions est aussi puissant que celui de DC.

On cherche à introduire un tel mécanisme en ARGOS. Les contraintes les plus simples, comme l'exclusion de deux signaux d'entrée, sont "sans mémoire". Elles ne portent que sur les variables d'entrée. On peut décider d'attacher à la définition d'un processus une formule booléenne qui définit les entrées valides. Un tel mécanisme est plus puissant que les deux types de contraintes d'ESTEREL, mais ne permet toujours pas d'exprimer des contraintes "à mémoire", comme l'alternance de deux signaux d'entrée.

On pourrait décider que la formule qui définit les entrées valides peut faire intervenir les états des automates du processus. Dans ce cas, l'un des automates, ou même une sous-expression du corps du processus, peut être entièrement consacré(e) à la définition des entrées valides. Pour exprimer que deux signaux d'entrée a et b alternent, on peut introduire un automate à deux états X et Y, qui n'émet rien. Il passe de X à Y sur occurrence de a, et de Y à X sur occurrence de b. Cet automate est placé en parallèle avec la structure qui définit le corps du processus proprement dit. La formule de définition des entrées valides s'écrit : $(a \implies X) \wedge (b \implies Y)$. Le choix de l'état initial, parmi X et Y, indique lequel des deux signaux doit commencer la séquence.

On peut aussi se contenter d'autoriser l'utilisation des signaux *locaux* ou même des signaux de *sortie* du processus, dans la formule de définition des entrées valides. C'est une manière détournée de faire référence à l'état, puisque les sorties et les signaux locaux dépendent de l'état. Dans l'exemple ci-dessus, on ajoute au processus le même automate à deux états, qui émet x (resp. y) continuellement quand il est dans l'état X (resp. Y). La formule se transforme en $(a \implies x) \wedge (b \implies y)$. Puisque les signaux x et y ne servent qu'à définir les entrées valides, ils ne sont pas visibles à l'extérieur : ce sont des signaux locaux attachés au processus.

Un autre moyen, calqué sur la définition des *observateurs* de programmes synchrones, consiste à prendre pour convention que la correction des entrées est définie par l'absence d'un signal dédié.

6.2 Aspects de syntaxe concrète

6.2.1 Abréviations des conditions booléennes

La première exigence concerne la définition de conventions raisonnables permettant d'abréger la notation des conditions booléennes de transitions.

On convient naturellement d'omettre les transitions qui bouclent et qui n'émettent pas de signal. Un programme peut donc toujours être complété, sans ambiguité, par les boucles manquantes. Pour chaque état X d'automate de base, la boucle manquante est étiquetée par $\neg(c_1 \vee c_2 \vee ... \vee c_n)$, où les c_i représentent les conditions des transitions issues de X.

6.2.2 Non déterminisme comme une abréviation

Nous nous plaçons dans le cas où le non-déterminisme intrinsèque des automates n'est pas accepté. Dans certains cas un non déterminisme apparent peut être interprété comme une abréviation. Si l'on écrit à partir d'un état X deux transitions dont les conditions sont a et b, l'automate est non déterministe, puisque $a \wedge b$ est un événement possible.

Plusieurs situations se présentent :

- On peut prouver qu'une des *contraintes sur l'environnement* implique que a et b sont deux signaux exclusifs, dans cet état X. Le non déterminisme n'est donc qu'apparent, et les

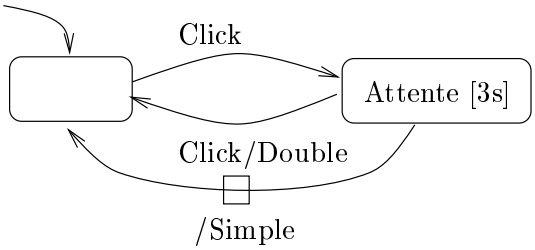


FIG. 6.1 – *Macro-notation de temporisation des états*

conditions pourraient être remplacées sans modification de sémantique par $a \wedge \neg b$ et $b \wedge \neg a$.

– Aucune contrainte explicitement spécifiée par le programmeur n'implique que a et b sont exclusifs dans X, ou bien on ne sait pas le montrer sans exploration de l'espace d'états. Toutefois la propriété est peut-être vraie, ou bien le programme global dans lequel l'automate considéré est utilisé assurera peut-être cette propriété, si a et b sont des signaux locaux. Au lieu de rejeter l'automate — et le programme — dès la phase d'analyse de déterminisme, on peut modifier localement le programme en ajoutant la contrainte d'exlusivité de a et b dans l'état X à la formule qui définit les entrées valides du processus. Le non-déterminisme des automates d'un processus est donc un moyen d'enrichir les contraintes d'environnement associées à ce processus.

6.2.3 Macro-notations

6.2.3.1 Etats temporisés

La notion d'état temporisé est définie au chapitre 10. Intuitivement, lorsque le système entre dans un état temporisé par le délai d sur le signal s, il ne peut y rester après que d occurrences du signal s ont été reçues. Une transition dite de "timeout" est prévue à cet effet. Le programme de la figure 6.1 permet de détecter les simples ou doubles clicks d'une souris.

L'expansion de la macro-notation de temporisation consiste à ajouter un automate compteur en parallèle avec la composante temporisée. Les deux composants communiquent dans les deux sens : le programme d'origine signale son entrée dans l'état temporisé, de manière à lancer le compteur ; il signale toute sortie normale de l'état temporisé, pour permettre de désarmer le compteur. Inversement, le compteur signale qu'il expire, et cela déclenche la transition de timeout.

Pour l'exemple de la figure 6.1 le programme expansé est donné par la figure 6.2.

6.2.3.2 Equations

L'idée d'introduire des équations est née de l'étude de la programmation multi-langages ARGOS+LUSTRE présentée au chapitre 8. Nous proposons ici une extension très simple de la syntaxe ARGOS, sous forme de macro-notation.

Une équation $x \leftarrow b$ où x est un signal de sortie, et b une formule booléenne quelconque faisant intervenir des signaux d'entrée et des signaux locaux, est directement utilisable, puisque ce n'est qu'une abréviation pour un automate à un état et une transition étiquetée par b/x, que l'on placerait en parallèle. On préfère la notation $x \leftarrow b$ à la notation $x = b$ de LUSTRE, pour

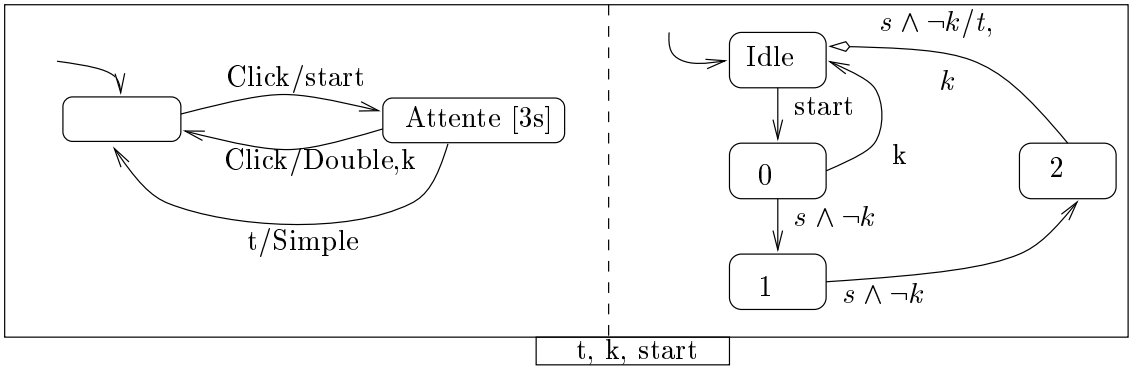


FIG. 6.2 – *Expansion de la macro-notation de temporisation*

ne pas avoir à exiger qu'un signal ne soit défini que par une seule équation. Un programme peut contenir deux équations $x \leftarrow b_1$ et $x \leftarrow b_2$, qu'on peut ou non remplacer par $x \leftarrow b_1 \vee b_2$.

6.2.3.3 Interruptions préemptives

Au chapitre 2 nous avons montré comment exprimer les interruptions préemptives grâce à une construction qui utilise le raffinement et l'opération de définition d'horloge (figure 2.10). Si l'on se donne un marquage des flèches de transitions pour repérer les transitions préemptives, il est aisé de produire un programme ARGOS de base, en ajoutant un signal d'horloge.

En syntaxe graphique, nous proposons une extension de ce genre au chapitre 9. En syntaxe textuelle, il faudrait trouver un mot clé et décider de l'endroit où il apparaît.

6.3 Un mécanisme général de pragmas

Les pragmas sont des informations non structurées — ou plutôt dont l'éventuelle structure n'est pas connue du compilateur ARGOS — et que l'on attache à certaines structures syntaxiques d'ARGOS. L'extension du compilateur prévoit d'attacher des pragmas aux états, aux transitions, aux processus, à la structure globale du programme, etc. L'application de la méthode au cas des systèmes hybrides ne nécessite que des pragmas d'états et de transitions.

Le compilateur a la charge de construire l'automate global, en transmettant les pragmas. La figure 6.3 donne un programme ARGOS équipé de pragmas, et la structure de l'automate global, équipé de pragmas composés.

Les pragmas associés aux états globaux héritent de la structure du programme ARGOS : ensemble de pragmas séparés par des virgules pour les compositions parallèles et un niveau de hiérarchie préfixé par le pragma de l'état raffiné, dans le cas des raffinements. Ainsi l'état XAC hérite-t-il du pragma composé : $[\chi\ [\alpha,\beta]]$.

Les pragmas associés aux transitions héritent également de la structure du programme. Le pragma associé à la transition globale de XAC vers Y (obtenu par composition des transitions $t1$, $t3$ et $t5$, conformément à la sémantique d'ARGOS) est : $[\chi\ [\tau_1,\tau_3],\tau_5]$.

Le compilateur transmet ainsi les pragmas en conservant l'information maximale sur la structure du programme, qui peut être nécessaire à l'utilisation desdits pragmas par un outil aval.

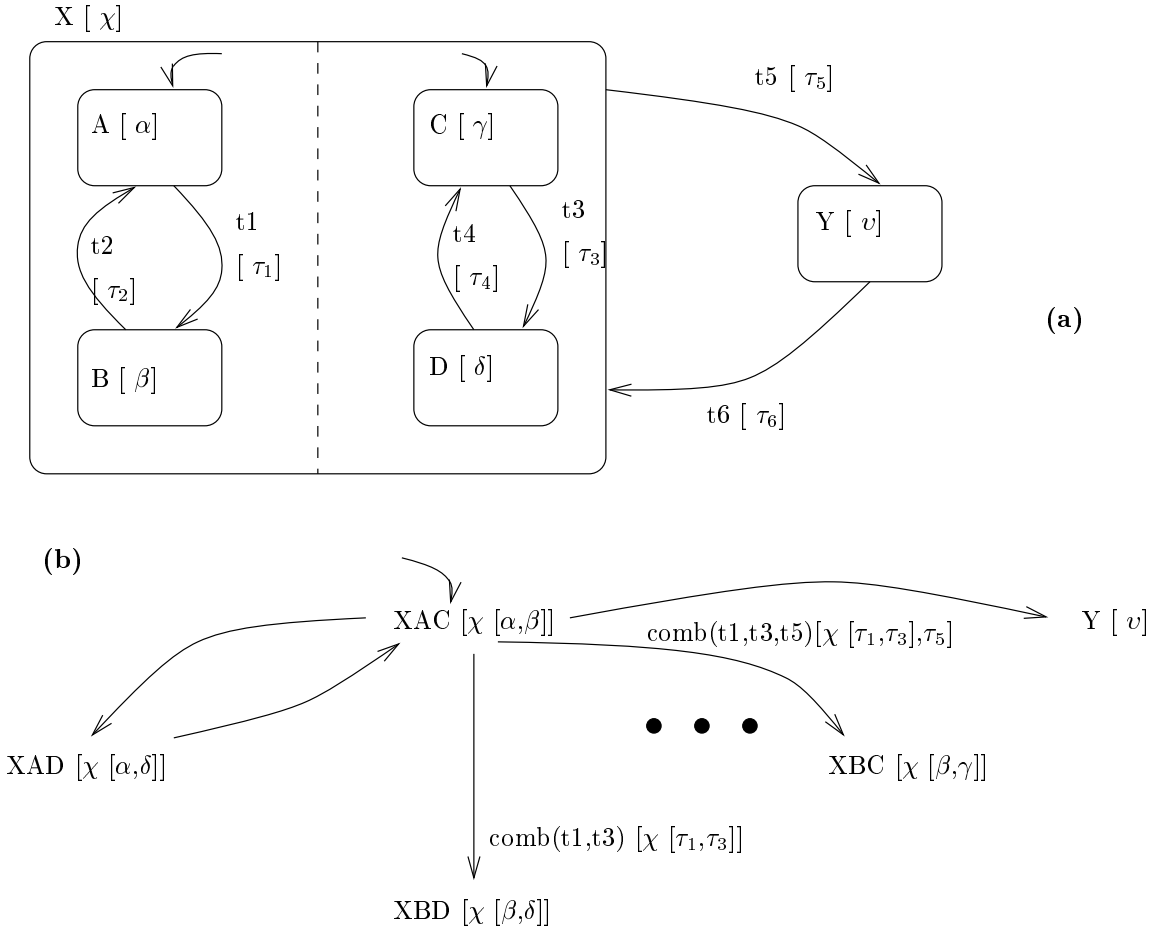


FIG. 6.3 – *Transmission des pragmas par le compilateur* ARGOS

Dans le cas de POLKA (voir chapitre 11 la connexion de l'environnement ARGONAUTE à l'outil POLKA, pour l'analyse des systèmes hybrides), les pragmas attachés aux états raffinés peuvent par exemple servir de déclarations de variables, locales aux pragmas du processus raffinant. Conserver la structure du programme ARGOS permet alors de définir des variables avec un classique mécanisme de blocs imbriqués.

Le mécanisme général des pragmas permet d'implanter les extensions temporelles et hybrides ou d'utiliser des variables données.

6.4 Extensions

6.4.1 Variables de contrôle non booléennes

Les variables de contrôle sont introduites pour éviter d'avoir à coder par un automate un élément du contrôle qui peut prendre un nombre fini et relativement petit de valeurs, mais pour lequel les transitions sont faciles à exprimer par des calculs. Typiquement, les automates compteurs introduits pour l'expansion de la macro-notation de temporisation seraient décrits beaucoup plus simplement avec des variables de contrôle qu'avec un automate explicite. La figure 6.4 donne un automate compteur sur l'intervalle $[0, 10]$, décrit en extension (a) et à l'aide d'une variable de contrôle notée c (b).

Les variables de contrôle sont du type intervalle d'entiers fini (on pourrait également envisager des types énumérés, mais alors on retrouve les automates explicites, puisque les transitions ne peuvent être décrites qu'en donnant explicitement tous les triplets <état source, étiquette, état but>), sur lequel on peut réaliser des opérations évaluables statiquement.

Si les variables de contrôle sont attachées à un processus, chacune d'entre elles représente un composant parallèle réduit à un automate, dont les états sont les valeurs possibles de la variable. Les conditions et affectations portant sur des variables de contrôle peuvent être explicitement traduites en communications par diffusion synchrone, à l'aide de signaux auxiliaires, ou directement interprétées par le compilateur. Dans tous les cas, les valeurs de la variable sont traitées comme des états du *contrôle*.

6.4.2 Variables "données" non booléennes

On peut vouloir également manipuler des variables *données*, c'est-à-dire dont les valeurs ne doivent pas faire partie de la structure de contrôle du processus. On peut définir un sous-langage d'ARGOS pour la définition et la manipulation de telles variables, éventuellement en important les types et les opérations depuis un langage hôte, comme c'est fait pour LUSTRE et ESTEREL. On peut aussi cacher entièrement ces aspects dans les pragmas.

6.5 Une proposition de syntaxe textuelle

La syntaxe textuelle présentée ici intègre les extensions nécessaires à la manipulation de l'horloge de base, à l'introduction de variables de contrôle, à la définition de pragmas, etc.

6.5.1 Variables de contrôle

Les variables de contrôle sont de type intervalle d'entiers. Les définitions des types utilisés et la déclaration des variables de contrôle sont attachés à la définition d'un processus. Ces variables

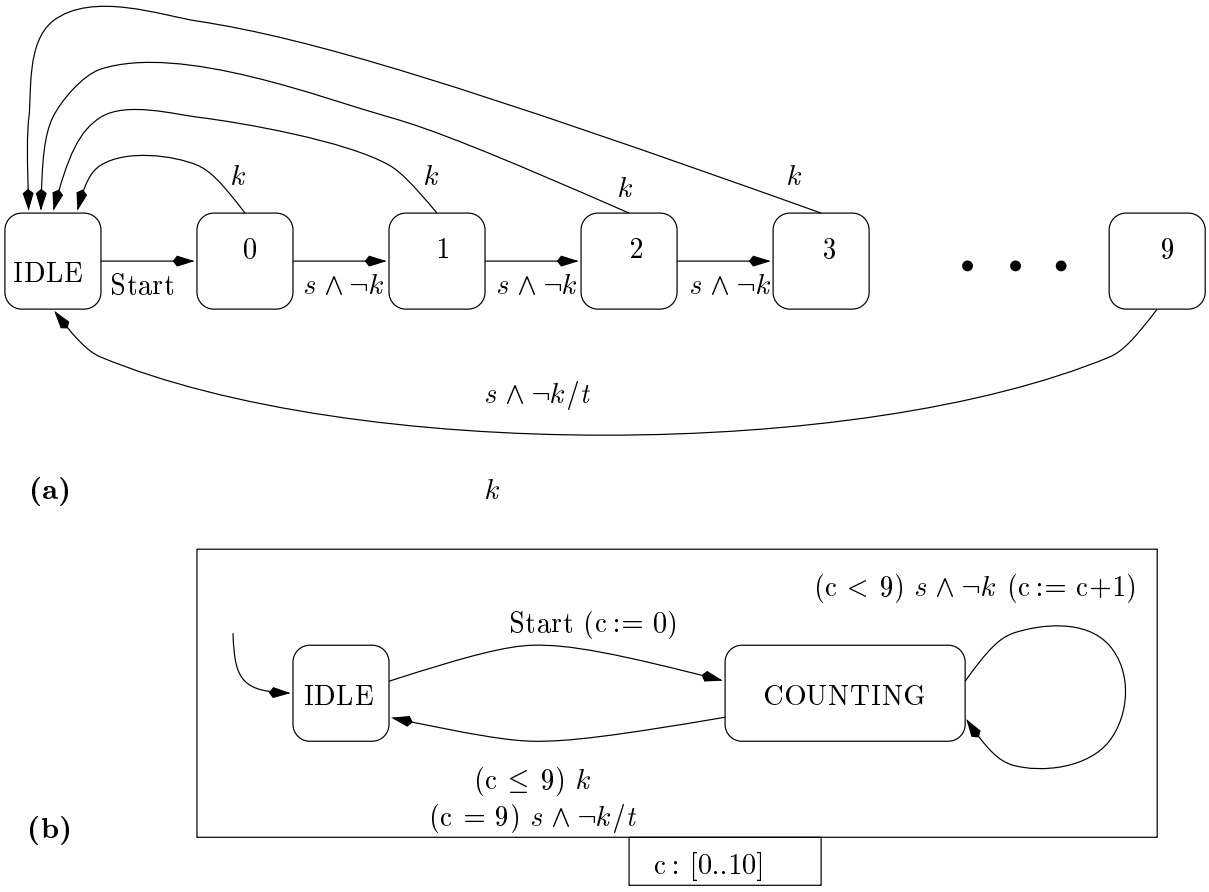


FIG. 6.4 – *Variables de contrôle ou automates explicites*

sont utilisables dans les transitions des automates de ce processus.

6.5.2 Etiquettes de transitions

Une étiquette de transition élémentaire comporte 4 champs :

- Une condition sur les variables de contrôle
- Une condition booléenne sur les signaux
- Un ensemble de signaux émis
- Une affectation aux variables de contrôle

Une étiquette est une suite d'étiquettes élémentaires séparées par des signes `+`. Cela permet de ne définir qu'une fois les états source et but de la transition, même si elle doit porter des quadruplets différents. Exemple : `(v = 5) & (v >3) u & x / z,t v := 2*3 + (v = 4) u/t`

6.5.3 Structure interne des processus

Les corps de processus sont décrits à l'aide des opérateurs suivants :

Composition parallèle, notée || La notation est infixée, l'opération est commutative et associative. `toto (x)(z) || tutu (y)(t) || titi (x)(z,t)` représente la composition parallèle de trois appels de processus.

Raffinement Le raffinement est considéré comme une structure de contrôle paramétrée par l'automate. Cette structure a donc n places d'arguments si l'automate de contrôle a n états. On note : `controller { ... }`. Le corps de cette structure est composé de la déclaration de chacun des états, suivi de son ensemble de transitions sortantes. L'état lui-même définit un bloc dans lequel on peut décrire le processus raffinant, s'il existe.

Encapsulation La question se pose de savoir si tous les signaux locaux d'un processus sont attachés à la définition du processus, et donc en quelque sorte déclarés à la racine de son arbre abstrait. Introduire un opérateur d'encapsulation (la structure `signal ... in ... end` d'ESTEREL) permet de déclarer des signaux locaux à des sous-arbres. Inversement, le choix d'attacher les déclarations à la définition de processus est analogue à celui de LUSTRE. Les auteurs de LUSTRE aimeraient maintenant en changer, il semble que ce soit parfois contraignant. Pour ARGOS, introduisons donc cet opérateur. La seule difficulté provient de la traduction en DC (cf. chapitre 5), dans laquelle on a décidé de construire un noeud DC par processus ARGOS. Or les signaux locaux des noeuds DC sont attachés à la définition du noeud. Il y a deux solutions : la première consiste à changer de méthode, et à produire un noeud DC par sous-arbre d'opérateur d'encapsulation ; la seconde consiste à "remonter" les déclarations de signaux locaux à la racine de l'arbre abstrait du processus, avec un éventuel renommage, mais il faut alors garantir que cela ne change pas la sémantique (voir les arguments dans ce sens, section 5.2.4 du chapitre 5).

Condition d'activation L'opération qui associe une condition d'activation à un processus est notée `when`. On écrit donc : `when B { P }` , où `B` est une condition booléenne et `P` un processus quelconque.

Notons que les automates sont des opérateurs. C'est une des raisons pour lesquelles les automates ne sont pas réutilisables comme les processus, par un mécanisme d'appel. Il faudrait plutôt un mécanisme d'instance de classe.

6.5.4 Contraintes sur l'environnement

Dans la syntaxe proposée ci-dessous les contraintes sur l'environnement sont attachées à la définition d'un processus, et ne portent que sur ses signaux d'entrée. On donne une formule booléenne quelconque de signaux d'entrée.

6.5.5 Temporisation des états

La borne de temporisation d'un état est optionnelle. Elle est associée à la définition d'un état : `state delay 4 ....`

6.5.6 Définition de processus, mécanisme d'appel et processus racine

A FAIRE

Le mécanisme d'étiquetage des appels de processus est introduit pour faciliter la remontée au source lors de l'expansion.

6.6 Exemple de programme

```
targos        // this is just a keyword...

/* Definition of the main process */

main P (a,b,c)(d,e)
with  (~a | b) &  #(a,b,c)  ;

/* Process declarations */

process P (x,y,z)(u,t)          // declares the name of the process, and
                                // the lists of formal input and output
                                // parameters.
[ this a pragma associated to process P ]
internal  v, w ;                // local signals.
types                           // types of control variables used in this process
    trois = 1..3     [ type pragma ] ;
    quatre = 1..4 ;
    cinq = 4..5 ;
var                             // CONTROL variables, with type and initial value.
    trois v = 2      [ variable pragma ] ;
    truc w = 1 ;
{                               // begin description of the process structure
internal a,b {
    tag1:Q (u)(k)               // tagged call of process Q
||
    controller {
           init One             // specifies the initial state
```

```
          // State One has two transitions, to states Two and Three.
          // and it is refined by a process made of Q, R and S.

          One [ this is a pragma associated to state One]
          {  when (x & ~y) {
                  tag1:Q (x)(t) ||
                  tag3:R (y)(t)  ||
                  tag4:S (x)(z,t)
             }
          }
          ->  Two             with u & ~x / z,t + ~u & x / z ;
          ->  Three           with y/t [ this a pragma for the transition ] ;

          // State Two is not refined. It has only one transition.
          // It has a delay attached to it.
          Two delay 4 {}      -> Two     with { (v = 5) & (v > 3) } u/c { v := 2*3 } ;

          // State Three is not refined. It has one transition.
          Three {}            -> One         with y/;
    } // end of "controller" structure
} // end of "internal" structure
} // end of process P

process Q (in)(out)  { ... }

process R (in)(out)  { ... }

process S (a)(b,c)     {  tag1:R (a)(b) || c <- a}

endtargos
```

Deuxième partie

Programmation multi-langages synchrone

Chapitre 7

La programmation multi-langages : motivations et première étude dans le cadre synchrone

Sources : *L'essentiel du travail sur le thème* programmation multi-langages *est décrit dans [JLMR94, JLMR93, Jou94]. On y trouve en particulier les motivations et le résumé d'une étude bibliographique débordant un peu du contexte des langages synchrones. Toutefois la première étude qui portait sur* ARGOS *et* LUSTRE *a soulevé de nouveaux problèmes, qui ont été traités indépendamment des aspects multi-langages. Nous reprenons ici l'énoncé des motivations, les propositions d'implantation décrites dans [JLMR93, Jou94], et les conclusions de ce travail, qui ont conduit à la proposition de mélanger les langages au niveau* DC. *La solution préconisée actuellement fait l'objet du chapitre suivant.*

7.1 Motivations et travaux sur le sujet

Les premières idées de programmation multi-langages sont anciennes dans la communauté synchrone, et proviennent de quelques observations simples. Les styles impératif textuel, impératif à base d'automates, flot de données déclaratif, etc., représentés parmi les langages synchrones, correspondent à des cultures différentes dans l'industrie. LUSTRE, surtout sous sa forme graphique, paraît ainsi très naturel dans une communauté qui utilise les schémas-blocs ; un langage à base d'automates rappelle le Grafcet aux concepteurs d'automates programmables. Toutefois, la description de systèmes de taille conséquente montre souvent les limites d'une approche homogène, quelle qu'elle soit. Dans une description flot de données, on peut avoir envie de décrire une structure de contrôle globale sous forme d'automate ; dans une description purement impérative, les fonctionnements réguliers qui impliquent beaucoup de calculs sont souvent lourds à décrire. En quelque sorte, le style déclaratif flot de données se prête bien à la description des régimes permanents, alors que les styles impératifs conviennent mieux à la description des régimes transitoires ou des ruptures de phases de fonctionnement. Un cas typique est constitué par les systèmes de vision par ordinateur, dans lesquels des phases de traitement du signal sont contrôlées par des modes de fonctionnement. De manière générale, tout système réactif de taille importante comprend les deux aspects. Les différents styles de programmation sont donc complémentaires plutôt que concurrents, et on aimerait pouvoir les utiliser conjointement dans une description de

grande taille.

D'autre part, l'approche synchrone permet l'expression des propriétés à vérifier par des *observateurs* [HLR93]. un observateur de la propriété P, pour un programme Q, est un autre programme O — qui peut donc être écrit dans le même langage — qui observe les sorties et les entrées de Q et n'a qu'une sortie: un booléen `ok`. On exécute Q et O en parallèle, et on observe la sortie `ok`. L'observateur est conçu de telle manière que `ok` est vraie tant que le programme Q vérifie la propriété P ; elle devient fausse, et le reste, dès que Q viole P. La programmation multi-styles ou multi-langages peut être utilisée dans ce cadre: le programme à vérifier peut par exemple être écrit en LUSTRE et l'observateur en ARGOS. Pour des propriétés du genre *au moins un 'a' dans chaque intervalle défini par les occurrences de 'b' et 'c'*, l'expression sous forme d'automate met clairement en évidence les cas limite, et peut-être plus lisible que la version LUSTRE. En suivant cette même idée, Pascal Raymond a proposé dans [Ray96] de décrire les propriétés de programmes synchrones par un langage à base d'expressions régulières, qui sont ensuite compilées en programmes LUSTRE grâce à l'outil REGLO.

Des préoccupations similaires apparaissent dans de nombreux contextes. Une étude bibliographique du sujet est nécessairement très partielle; l'étude des langages est au coeur de l'informatique, et le thème de la programmation *multi-styles*[1] est un thème transversal à tous les domaines de l'informatique. Une recherche rapide dans la *Collection of Computer Science Bibliographies*, pour le mot-clé *multiparadigm*, donne des références relatives aux langages visuels, à la programmation concurrente orientée objet, à la spécification de systèmes téléphoniques, à l'ordonnancement de tâches, à la construction de compilateurs, à la modélisation du processus de développement et, bien sûr, à la recherche d'informations (!).

Toutefois nous n'avons pas trouvé de travaux sur le thème de la programmation multi-styles en soi, indépendamment d'une d'application particulière. Les travaux les plus accessibles concernent des langages généraux, et proposent de mêler les styles objet, logique, impératif, ... Voir par exemple la définition de LEDA, un langage qui offre les styles impératif, fonctionnel, orienté objet et relationnel [Shu91, PPST93], ou celle de Oz [Smo94b, Smo94a] (plus de détails à l'adresse: `http://ps-www.dfki.uni-sb.de/oz/`). D'après ses auteurs, Oz est basé sur un nouveau modèle de calcul qui permet d'intégrer simplement les styles fonctionnel d'ordre supérieur, logique avec contraintes et orienté-objet concurrent. Oz est conçu comme le successeur de Lisp, Prolog et Smalltalk. L'accent y est mis sur la définition d'un petit ensemble de constructions sémantiques, à partir desquelles peuvent être dérivées des constructions usuelles des langages orienté objet, par exemple.

7.2 Programmation multi-langages dans le cadre synchrone

Notre point de vue est moins général, puisque nous nous intéressons au problème relativement précis de la programmation *multi-langages*, à partir d'un petit ensemble de langages bien identifiés, qui disposent en outre d'une base sémantique commune. J'utilise l'adjectif "multi-langages" pour des approches qui consistent à considérer des programmes écrits dans des langages différents, et à définir une sémantique qui étend chacune des sémantiques d'origine. Une approche multi-langages est plus intéressante si elle porte sur des langages de styles différents.

Dans [JLMR94], nous proposions une approche de programmation multi-langages *au niveau source*, par opposition à l'utilisation d'une technique d'éditions de liens. Une réflexion plus appro-

1. L'approche est en général qualifiée de "*multi-paradigmes*". Bien que le terme semble consacré, il est fort peu approprié en français, et nous lui préférerons celui de "multi-styles".

fondie montre que la différence entre ces deux approches n'est pas fondamentale ; ce n'est qu'une question de degré. L'approche de compilation d'un programme mixte ARGOS+LUSTRE via le format DC, que nous proposons au chapitre 8, suit exactement le même schéma que la compilation d'un programme écrit en C et PASCAL, via le format des fichiers binaires translatables.

7.2.1 Travaux de même nature

Dans la communauté synchrone ou à sa périphérie, on trouve le système PTOLEMY développé dans l'équipe d'Edward Lee à Berkeley. PTOLEMY est un environnement de programmation pour le développement, la simulation, et le prototypage de systèmes de traitement du signal et temps réel. PTOLEMY offre différents modèles d'exécution qui permettent de spécifier les interactions entre les calculs et leur ordonnancement. En outre, PTOLEMY permet de mélanger plusieurs modèles d'exécution dans une même application afin de choisir le modèle le mieux adapté à chaque sous-système. Parmi les modèles d'exécution on trouve un mode qualifié de "flot de données synchrone" et un mode "machine d'états finie". Les problèmes posés sont toutefois d'un autre ordre que ceux auxquels nous nous intéressons ici : PTOLEMY cherche à intégrer, non seulement des styles de programmation différents, mais aussi des modèles de calcul différents.

Le travail présenté ici s'apparente plus à [ZJ93], que P. Zave m'a signalé depuis à l'occasion d'un exposé sur la programmation ARGOS+LUSTRE via DC (objet du chapitre 8). On trouve également dans [CV89] une proposition de combinaison des statecharts et de la logique temporelle PTL pour la spécification des systèmes réactifs, où les Statecharts sont utilisés pour décrire des comportements trop généraux, qui sont ensuite restreints par des propriétés exprimées en PTL.

Une équipe des Bell Labs s'intéresse à la combinaison ESTEREL+ML.

Avec l'objectif de fournir en LUSTRE des structures de contrôle permettant par exemple de ré-initialiser un processus, Paul Caspi et Marc Pouzet ont proposé une extension de LUSTRE par ajout de structures récursives. [RM95] décrit d'autre part une proposition d'introduction de structures de contrôle dans SIGNAL, sous formes d'intervalles d'activation des processus. Cette extension a été utilisée pour décrire un système de vision.

7.2.2 Analogie avec C et PASCAL

Nous concluions dans [JLMR94] que la programmation multi-langages n'est envisageable que si l'on sait identifier, dans chacun des langages candidats, un *niveau d'encapsulation* adéquat, qui définit les blocs échangeables. Dans le cas de C et PASCAL, il s'agit des procédures ou fonctions. On peut appeler une fonction C depuis une procédure PASCAL, parce que la sémantique de ces blocs est comparable (suffisamment, en tout cas, pour que les deux objets soient compilés de la même façon, et que les fichiers objets soient fusionnables — les conventions d'utilisation de la pile ou des mécanismes de fenêtres de registres pour le passage de paramètres étant imposées par le système, et respectées par les différents compilateurs). On ne peut guère descendre plus bas que la structure de procédure ou fonction pour mêler C et PASCAL : dès que l'on imagine des imbrications de structures de contrôle, on rencontre des programmes comme :

```
while x < y do  (* c'est du pascal *)
begin
     if ( f(x,y)) break ;       /* c'est du C */
end ;
```

où des instructions qui n'ont de sens que dans un contexte C apparaissent dans un contexte PASCAL.

La relation est évidente entre cette notion de bloc échangeable, et la notion de *programme*, par opposition aux *termes*, utilisée pour la formulation de la propriété de complète adéquation d'une sémantique (voir chapitre 2, section 2.5.2).

Le seul argument pertinent pour distinguer un mélange *au niveau source* concerne les aspects pratiques d'analyse des fichiers source : peut-on réellement mêler les constructions de deux langages dans un même fichier source, auquel cas on doit imaginer un analyseur syntaxique capable de comprendre un sur-ensemble des deux syntaxes ? Mais il y a de fortes chances pour que le niveau de bloc que l'on a identifié comme adéquat pour l'échange entre langages, soit également le niveau de bloc utilisé pour la programmation modulaire, dans chacun des deux langages pris séparément. On peut donc fournir les fragments de programmes dans des fichiers distincts, en confier l'analyse aux analyseurs mono-langages, et ne s'occuper que de la fusion des objets compilés.

7.2.3 Le cas ARGOS+LUSTRE

Dans le cas des langages synchrones, et en particulier de LUSTRE et ARGOS, le problème est à la fois beaucoup plus facile, un peu plus difficile et beaucoup plus intéressant, que dans le cas de C et PASCAL. Difficile et intéressant, parce que les structures d'ARGOS et de LUSTRE sont bien plus différentes que celles de C et PASCAL ; facile parce que la base sémantique commune est beaucoup plus large (et formellement définie).

Comprendre comment les deux langages peuvent être mêlés, et estimer la richesse du mélange obtenu, suppose donc que l'on clarifie cette base sémantique commune. C'est ce que nous montrions dans [JLMR94], en présentant des abstractions de LUSTRE et ARGOS, nommées respectivement L et A. Ces abstractions permettent de comprendre comment l'on décrit des objets de base et comment l'on compose des objets, dans chacun des deux langages. On montre ensuite que les objets de base décrivent les mêmes objets sémantiques, puis on propose un langage mixte qui offre : les deux façons de décrire des objets de base, toutes les manières de composer des objets.

On peut en donner une vue encore plus abstraite, en oubliant momentanément le raffinement (la formulation qui suit est inspirée par des discussions avec Paul Caspi, et par ses présentations des fondements de l'approche synchrone).

Considérons un ensemble de signaux d'entrée I et un ensemble de signaux de sortie O. Un comportement réactif peut être décrit de deux manières : a) par un système de transitions fini — une machine de Mealy comme définie au chapitre 2 ; b) par une fonction f de $P(I)^*$ (les séquences d'entrée) vers $P(O)^*$ (les séquences de sortie), à mémoire bornée. Les deux modèles sont interchangeables. En effet, si f n'utilise qu'une abstraction bornée de la séquence d'entrées pour calculer la séquence de sorties, elle peut être représentée par un système de transitions dont les états correspondent aux valeurs possibles de la mémoire. Inversement, tout système de transitions fini représente une fonction des séquences d'entrées vers les séquences de sorties.

ARGOS peut être vu comme l'exemple type le plus simple d'un langage basé sur le point a) ci-dessus : les objets de base sont donnés par des systèmes de transitions et la composition repose sur le produit synchrone. Les programmes sont définis par :

$$\begin{aligned} \text{A} &::= \text{BasicA} \mid \text{A}\|\text{A} \\ \text{BasicA} &::= \ldots \end{aligned}$$

En ARGOS on donne explicitement les ensembles d'états et de transitions des composants de base. C'est un héritage de la syntaxe graphique, mais on peut imaginer d'autres genres de descriptions

pour les BasicA. La sémantique est donnée par une fonction : $\mathcal{S}_i$: A $\longrightarrow \mathcal{M}$ (où $\mathcal{M}$ est l'ensemble des machines de Mealy définies au chapitre 2), qui vérifie : $\mathcal{S}_i(\mathrm{A}_1 \| \mathrm{A}_2) = \mathcal{S}_i(\mathrm{A}_1) \times \mathcal{S}_i(\mathrm{A}_2)$.

LUSTRE est l'exemple type d'un langage basé sur le point b). Les objets de base sont des fonctions décrites par des systèmes d'équations. Ces objets sont connectés par des "fils", pour former des réseaux flot de données, qui peuvent être décrits graphiquement ou textuellement, par des systèmes d'équations (on nomme les fils par des variables). Les programmes peuvent être décrits par :

$$\begin{array}{rcl} \mathrm{DF} & ::= & \mathrm{BasicDF} \mid \{\mathrm{eq}\} \\ \mathrm{eq} & ::= & o = \mathrm{BasicDF}(i) \end{array}$$

où o et i sont des n-uplets de noms de variables, et représentent les séquences de valeurs qui passent dans les fils. Les BasicDF sont les fonctions à mémoire bornée qui forment les "noeuds" du graphe flot de données. En LUSTRE on garantit que ces fonctions sont à mémoire bornée par des restrictions syntaxiques : les équations qui les définissent sont de la forme :

$$\mathrm{BasicDF} \quad ::= \quad \mathrm{Inst} \mid \mathrm{pre}$$

où Inst dénote des fonctions instantanées, ou combinatoires, et Pre est l'unique moyen de faire référence au passé. Les fonctions combinatoires sont obtenues comme des extensions aux séquences de fonctions point à point. Par exemple, $f(x) = 2 \times x$ est étendue en une fonction f telle que : $o = f(i) \iff \forall k \geq 0, o_k = 2 \times i_k$. La fonction pre est définie par : $\forall k > 0, o_k = i_{k-1}$. La fonction sémantique $\mathcal{S}_{df}$: DF $\longrightarrow \mathcal{M}$ vérifie :

$$\mathcal{S}_{df}(\{o = B(i), o' = B'(i'), ...\}) = \mathcal{S}_{df}(B) \times \mathcal{S}_{df}(B') \times ...$$

Dans le langage mixte, on autorise la description de comportements de base par des systèmes de transitions ou par des équations décrivant des fonctions à mémoire bornée ; on permet la composition par le produit synchrone ou par la connexion en réseau d'opérateurs. D'où la grammaire :

$$\begin{array}{rcl} \mathrm{Mixed} & ::= & \mathrm{MixedBasic} \mid \mathrm{Mixed} \| \mathrm{Mixed} \mid \{\mathrm{eq}\} \\ \mathrm{MixedBasic} & ::= & \mathrm{BasicA} \mid \mathrm{BasicDF} \\ \mathrm{eq} & ::= & o = \mathrm{MixedBasic}(i) \end{array}$$

On fusionne aisément les deux sémantiques.

7.2.4 Implantation

Lorsque l'on s'intéresse à une utilisation pratique de ces idées simples, il faut tout de même résoudre quelques petits problèmes. ARGOS, et surtout LUSTRE, ne sont pas aussi simples que cette présentation abstraite pourrait le faire croire. Nous proposions dans [JLMR94, JLMR93] un sous-ensemble raisonnable de LUSTRE qui se prête sans difficulté à l'implantation des idées exposées ci-dessus. Nous montrions comment compiler le langage mixte en deux étapes : une traduction des fragments ARGOS en LUSTRE, accompagnée d'une modification structurelle des fragments LUSTRE, suivies de la compilation LUSTRE usuelle.

La modification des fragments LUSTRE n'est pas mentionnée dans la présentation abstraite ci-dessus, pour les besoins de laquelle nous avions oublié le raffinement. Elle consiste à rendre les noeuds "*ré-initialisables*", pour pouvoir les utiliser dans un raffinement ARGOS. Cela consiste à rajouter à chaque noeud LUSTRE deux signaux d'entrée booléens qui déterminent l'état d'activité

du noeud pendant et après la réaction courante. Cette opération est décrite au chapitre 5 à propos de la traduction du raffinement ARGOS en équations booléennes, je ne la détaille donc pas ici.

La traduction structurelle d'ARGOS en LUSTRE pose le problème de l'analyse de causalité des fragments ARGOS, avant (ou pendant) la traduction en LUSTRE. Ce problème n'était toujours pas réglé dans [Jou94]. L'étude d'une traduction d'ARGOS en équations booléennes sans contraintes (c'est-à-dire autorisant des cycles de définitions), et l'utilisation de l'outil BAC (voir la section *Analyse de causalité sur la forme équationnelle*, au chapitre 2) ont permis de mieux cerner les problèmes. Le chapitre 8 expose les problèmes et montre comment intégrer une analyse de causalité des fragments ARGOS dans un schéma d'implémentation où LUSTRE et ARGOS sont traduits en DC. Nous envisageons maintenant une véritable programmation ARGOS+LUSTRE, en utilisant LUSTRE sans restrictions et en garantissant aux fragments ARGOS une analyse de causalité complète.

7.3 Conclusions

L'application idéale des idées exposées ici serait un environnement de programmation et vérification des systèmes réactifs dans lequel les programmes pourraient être décrits indifféremment en LUSTRE, ARGOS, ESTEREL, REGLO, etc.

En attendant qu'un tel objet soit effectivement réalisé, les études menées sur la programmation multi-langages dans des cas particuliers — mélange ARGOS+LUSTRE décrit au chapitre 8, mélange ARGOS+ESTEREL décrit au chapitre 9 — peuvent être utilisées pour imaginer l'enrichissement d'un langage par des structures issues d'un autre. La première idée de ce genre, issue de l'étude ARGOS+LUSTRE, est d'introduire des équations en ARGOS. La deuxième idée provient de l'étude ARGOS+ESTEREL, et consiste à offrir une notation pour les interruptions préemptives. Ces extensions sont présentées au chapitre 6.

Chapitre 8

Argos + Lustre via DC

Sources : *La traduction d'*ARGOS *en systèmes d'équations booléennes avec conditions d'activations, publiée dans [MH96a], est déjà présentée dans la première partie (chapitre 5). Il reste à exposer les problèmes d'analyse de déterminisme et réactivité, la ré-initialisation des noeuds* DC *issus de* LUSTRE *(qui fait l'objet du mémoire de DEA de Traian Popovici) et le mécanisme de fusion des fichiers* DC *issus des compilateurs* ARGOS *et* LUSTRE *(ce dernier point constitue le sujet de magistère de Yann Rémond).*

8.1 Structure des programmes et exemple

Dans [MH96a] nous avons réduit DC à sa plus simple expression. En réalité il s'agit d'un format assez riche, qui offre par exemple une notion de noeud analogue à celle de LUSTRE pour la structuration des programmes. Nous considérons également une syntaxe structurée d'ARGOS, qui offre la notion de *processus*. La traduction choisie est telle que l'on crée un noeud DC pour chaque processus ARGOS. La liaison entre les deux langages repose sur le mécanisme d'appel de noeud. Tout processus ARGOS peut être construit en utilisant un noeud défini dans le fichier LUSTRE. Inversement, tout noeud LUSTRE peut faire appel à un processus ARGOS. Chacun des fichiers est analysé par le compilateur adéquat, et l'on produit un ensemble de noeuds DC correspondant aux processus ARGOS, et un ensemble de noeuds DC correspondant aux noeuds LUSTRE. Le programme global doit bien sûr contenir tous ces noeuds.

La figure 8.1 donne un exemple très simple de programme mixte, dans lequel la structure principale est décrite en ARGOS. L'un des composants est un noeud LUSTRE, défini ci-dessous :

```
node L (x : bool) returns (y,z : bool)
var M : bool ;
let
    M = x ->not pre(M) and x ;
    y,z = A2 (M, x) ;
tel.
```

Lequel, lui-même, fait appel à un processus ARGOS `A2` également donné figure 8.1.
La figure 8.2 donne l'arbre abstrait de ce programme, en montrant les sous-arbres LUSTRE et les sous-arbres ARGOS.

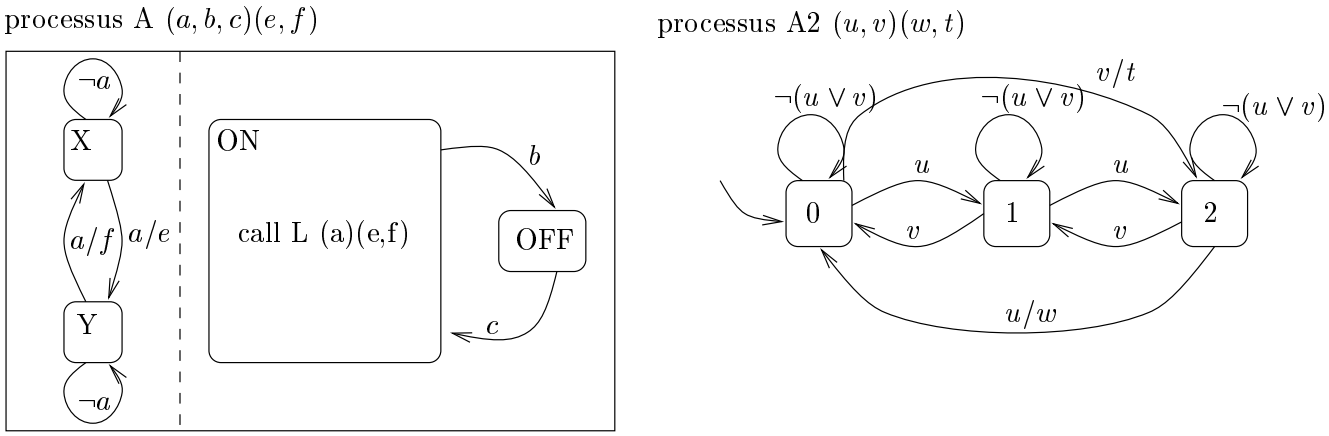


FIG. 8.1 – *Un programme mixte* LUSTRE+ARGOS *(partie* ARGOS*)*

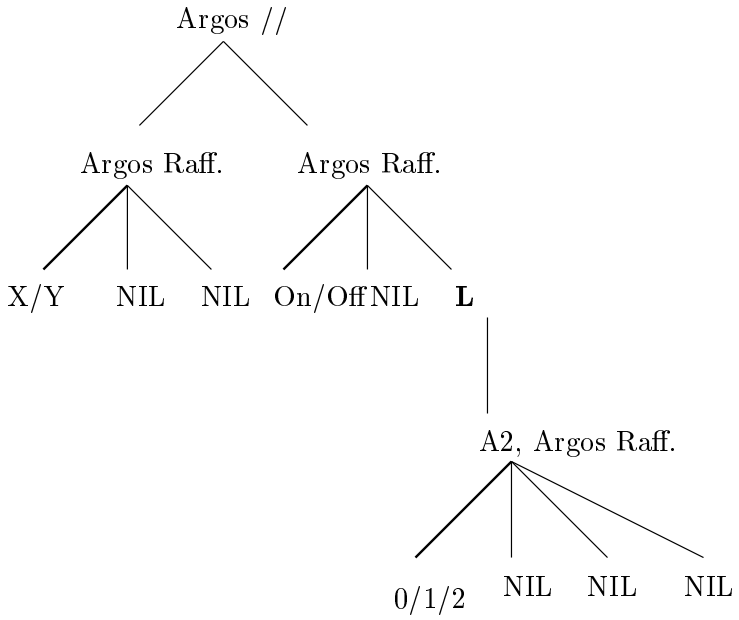


FIG. 8.2 – *Arbre abstrait d'un programme mixte*

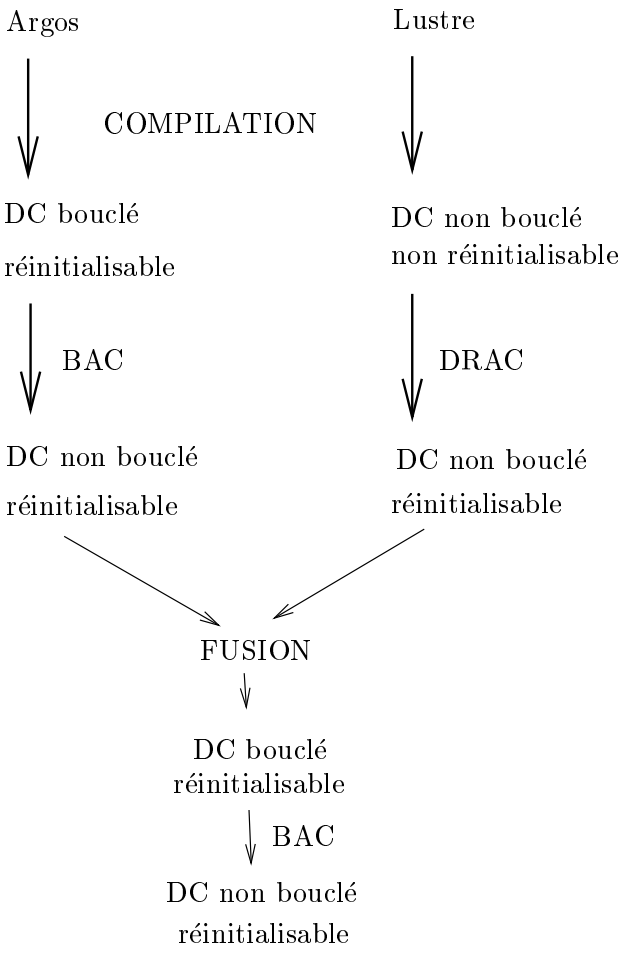


FIG. 8.3 – *Programmation mixte* ARGOS+LUSTRE *via* DC

8.2 Schéma de principe

La figure 8.3 donne le schéma de principe d'un environnement de programmation mixte ARGOS+LUSTRE, avec fusion au niveau DC. Le compilateur ARGOS produit naturellement des noeuds DC réinitialisables (cf. chapitre 5, il suffit de conserver les attributs hérités kill et alive comme paramètres de chaque noeud DC produit pour un processus ARGOS). Cette propriété permet d'intégrer le noeud dans un contexte comme le raffinement ARGOS, où il peut être lancé, interrompu, relancé dans son état initial. Le compilateur LUSTRE, en revanche, produit des noeuds DC non réinitialisables.

D'autre part, si le compilateur LUSTRE garantit que les ensembles d'équations des noeuds DC produits sont sans boucle, ce n'est pas le cas pour le compilateur ARGOS.

On introduit donc deux outils de modification des fichiers au format DC: un "*déboucleur*", et un outil de réinitialisation. Ces outils sont utilisés avant fusion, pour ne fusionner que des ensembles de noeuds à la fois non bouclés et réinitialisables. La fusion risque toutefois de réintroduire des boucles (cf. ci-dessous). On doit donc appliquer le mécanisme de débouclage une fois encore.

8.3 Outil de débouclage des noeuds DC issus d'ARGOS

A l'échelle d'un noeud DC produit par compilation d'un processus ARGOS, on peut rassembler toutes les déclarations de signaux locaux (la syntaxe proposée au chapitre 6 ressemble d'ailleurs à celle de LUSTRE, où la déclaration des signaux locaux est toujours attachée à la définition d'un noeud). On peut donc considérer que le processus ARGOS n'a qu'une opération d'encapsulation, située à la racine.

On adapte la technique de l'outil BAC décrite au chapitre 5, paragraphe 5.2, qui produit : un ensemble d'équations non bouclées, et une condition nécessaire et suffisante portant sur les variables d'état et les signaux d'entrée, qui détermine dans quelles configurations le programme est à la fois déterministe et réactif. Cette condition est ajoutée à la partie *assertion* du noeud DC. Le débouclage étant nécessaire à chaque encapsulation, on l'applique à tout noeud DC issu d'ARGOS.

8.4 Réinitialisation des noeuds DC issus de LUSTRE

Pour être utilisé dans un raffinement ARGOS, un programme LUSTRE doit être rendu *réinitialisable*.

La transformation consiste à ajouter deux paramètres d'entrée kill et alive, avec la même signification qu'au chapitre 5 : kill signifie que le processus considéré est tué durant la réaction courante ; alive signifie qu'il participe à la réaction courante. On effectue ensuite les trois modifications suivantes :

- Ajouter le paramètre alive à toutes les conditions d'activation. Ainsi le noeud considéré ne calcule rien aux instants où alive est faux, s'il est placé dans un contexte qui peut l'activer et le désactiver.

- Ajouter une nouvelle variable locale reset définie par une mémorisation :
  MEM($reset, kill$, TRUE, FALSE).
  reset est vrai lorsque kill était vrai à l'instant précédent. reset détermine des instants où tous les objets doivent prendre leurs valeurs initiales.

- Modifier toutes les parties droites des définitions d'origine en tenant compte de kill et reset.

D'où :

Equations : En DC, la valeur x_0 des équations est utilisée au début, avant que le premier instant où C est vrai permette de définir x en calculant E. x vaut donc x_0 jusqu'au premier instant où C est vrai *exclu*. Dans un noeud DC réinitialisable, la valeur x_0 est utilisée dans les intervalles définis par une occurrence de reset et une occurrence de C. Lorsque C et reset sont simultanés, l'intervalle est vide et on n'a pas besoin de x_0, x est tout de suite calculé d'après E. L'essentiel est de remplacer le recours à la valeur précédente de x par x_0, lorsque le noeud est réinitialisé. Une équation EQU(x, E, C, x_0) est donc remplacée par EQU$(x,$ if C then E else x_0, C $\wedge$ alive $\vee$ reset, $x_0)$.

Mémorisations : En DC, la valeur x_0 des mémorisations est utilisée comme valeur intiale, avant toute affectation à x. x vaut donc x_0 jusqu'au premier instant où C est vrai *inclus*. Dans un noeud DC réinitialisable, la valeur x_0 est utilisée dans les intervalles définis par une occurrence de reset et l'instant suivant une occurrence de C. Une mémorisation MEM(x, E, C, x_0)

est donc remplacée par
MEM(x, if kill then x_0 else E, kill $\vee$ C $\wedge$ alive, x_0)

8.5 Un mécanisme d'édition de liens au niveau DC

L'édition de liens des ensembles de noeuds DC ayant subi les modifications ci-dessus consiste simplement à rassembler tous les noeuds. Pour que l'ensemble constitue un programme exécutable, il ne doit plus y avoir de noeud indéfini. D'autre part la structure des appels de noeuds ne doit pas présenter de cycle.

Enfin, et c'est le plus ardu, il peut apparaître de nouveaux cycles de définition de signaux, lors de la fusion des deux ensembles de noeuds. Il faut donc réaliser globalement une phase de débouclage.

8.6 Travaux de même nature

L'équipe d'Axel Poigné à la GMD (Sankt Augustin) travaille sur le projet Synchronie, dont l'un des objectifs est de définir et implanter un environnement de programmation synchrone basé sur LUSTRE, ESTEREL et ARGOS. Les membres de l'équipe ont pour cela réécrit la sémantique des trois langages en termes d'*automates booléens*, un format équationnel de définition d'automates, assez similaire à DC. La fusion des langages est réalisée au niveau des automates booléens. Cette réécriture des sémantiques s'accompagne de quelques modifications de fond, comme la suppression des variables en ESTEREL.

Ayant commencé avec des ambitions assez vastes — comme par exemple de réaliser une imbrication plus fine d'ARGOS et LUSTRE, dans laquelle les transitions ARGOS auraient porté des portions de programmes LUSTRE — ils semblent maintenant s'être ramenés à des choix semblables aux nôtres. Il semble pourtant difficile de profiter de leur travail, même en partie, puisque les formats, les langages et les compilateurs utilisés sont entièrement redéfinis.

8.7 Pour aller plus loin...

La technique de mélange de langages par un mécanisme d'édition de liens au niveau DC peut être poussée plus loin. On peut ainsi envisager de mêler ARGOS, LUSTRE et ESTEREL dans un schéma similaire à celui décrit plus haut. Un noeud DC issu des compilateurs LUSTRE ou ARGOS doit être équipé d'entrées et de sorties spécialisées, pour intégration dans un contexte ESTEREL quelconque. Même si l'on restreint ces contextes ESTEREL à des contextes d'appel de *modules* (c'est-à-dire sans récupération des exceptions), il faut prévoir 4 entrées go, resume, suspend et kill et deux sorties halt et term. Par exemple, le processus se réincarne lorsque go et res sont tous deux présents. Le contexte permet de déterminer le degré de réincarnation (c'est-à-dire le nombre d'instances simultanément vivantes) du processus que l'on placera dans ce contexte. Un noeud DC candidat à occuper ce contexte doit être modifié pour présenter autant d'exemplaires que nécessaire de ses signaux d'entrée.

Une version simplifiée pourrait consister à n'autoriser l'intégration dans un contexte ESTEREL de noeuds DC issus des autres compilateurs, qu'à la condition que le contexte soit sans réincarnation, ce qui peut être déterminé statiquement.

Chapitre 9

Argos + Esterel

Sources : *Les propositions pour un langage mixte* ARGOS+ESTEREL *présentées ici sont issues de discussions avec Gérard Berry et Nicolas Halbwachs, après que G. Berry eut fait état de préoccupations relatives à la syntaxe chez les utilisateurs d'*ESTEREL *(paragraphe "Motivations"). La version correcte des schémas de traduction structurelle est due en grande partie à N. Halbwachs. Ce travail n'est pas publié dans son intégralité. Il n'existe pour l'instant que sous forme de rapport de recherche embryonnaire, l'existence de travaux tout à fait similaires et plus accomplis (dernier paragraphe) nous ayant dissuadés de poursuivre la réflexion jusqu'à l'implantation.*

Le dernier paragraphe est tiré de réflexions tout à fait personnelles autour de la notion de causalité. L'interprétation de cette notion, et les conclusions qui en découlent, sont parfaitement indépendantes des motivations et de la traduction proposée.

9.1 Motivations

Le mélange ARGOS+ESTEREL répond bien sûr aux motivations générales exposées au chapitre 7, complétées par ce qui suit :

Du point de vue d'ESTEREL **:** Dans un programme ESTEREL, les structures de contrôle temporelles comme le chien de garde (`watching`) et la détection d'exception (`trap`) définissent des contextes de préemption imbriqués. La syntaxe textuelle d'ESTEREL ne montre pas toujours assez explicitement les interactions entre une suite d'instructions et son contexte de préemption. Dans les langages séquentiels, il est possible de comprendre le corps d'une itération en s'abstrayant de la condition de continuation. En revanche, un chien de garde ESTEREL interagit avec son corps *à tout instant* de l'exécution de celui-ci. La compréhension d'une portion de programme exige donc la lecture de toutes les structures de contrôle englobantes, qui peuvent être assez éloignées dans le texte, et dont l'imbrication n'est manifeste que dans l'indentation. Une syntaxe graphique pourrait peut-être rendre la structure des programmes plus lisible.

D'autre part, dans un contexte de description impérative des systèmes réactifs, on rencontre souvent des systèmes dont la description la plus naturelle est donnée sous forme d'automate. Il n'y a pas de moyen simple pour décrire une telle machine, de manière lisible, avec les structures d'ESTEREL. La traduction systématique d'un automate en programme ESTEREL, proposée plus loin, permet de s'en convaincre.

Du point de vue d'ARGOS**:** Les syntaxes graphiques doivent se contenter d'un nombre assez restreint de caractères (formes, couleurs, positions) significatifs. Une syntaxe graphique devient rapidement illisible par accumulation de caractères. Dans le cas d'ARGOS, il ne serait guère raisonnable par exemple de distinguer des états carrés ou rectangulaires, à coins carrés ou arrondis, à trait épais ou fin, etc. Le nombre de constructions syntaxiquement distinguables est donc réduit. D'autre part la taille d'un écran ou d'une feuille de papier impose des limitations de taille aux portions de programmes qui doivent être lues d'un seul bloc, et il s'avère en général que, pour la même limitation de taille, une syntaxe graphique offre moins d'information qu'une syntaxe textuelle.

Toutes ces raisons font qu'un programme ARGOS de moyenne complexité est assez encombrant (jeu de construction réduit et syntaxe peu concise). D'autre part l'utilisation des deux dimensions que permet une syntaxe graphique est intéressante quand il s'agit de visualiser les structures de contrôle emboîtées qui constituent l'architecture globale d'un programme, mais dans la description des détails, lorsqu'il s'agit finalement de décrire des enchaînements d'actions simples, on aimerait se ramener à une syntaxe textuelle plus économique. Enfin les constructions d'ARGOS représentent un sous-ensemble des primitives d'ESTEREL, et même si les primitives manquantes peuvent être reconstruites, cela rend les programmes peu concis et moins lisibles. Mêler des portions de code ESTEREL aux programmes ARGOS pourrait apporter une solution à ces problèmes.

9.2 Proposition : ARGOS+ESTEREL

Du point de vue langage, notre proposition peut être vue de deux manières : 1) comme une extension d'ARGOS par quelques structures d'ESTEREL permettant de réduire la taille des programmes, dans des cas bien connus ; 2) comme l'ajout à ESTEREL d'une structure de contrôle temporelle supplémentaire basée sur l'idée du raffinement ARGOS. D'un point de vue pratique, nous proposons de traduire le langage mixte en ESTEREL, pour bénéficier en aval de tout l'environnement de programmation ESTEREL (génération de code, simulation, débogage symbolique, systèmes de validation, ...).

Les propositions portent sur des extensions indépendantes les unes des autres, parmi lesquelles il devrait être possible de choisir librement. Nous étudions toutefois les interactions entre extensions, là où elles risquent de rendre le langage incompréhensible ou ambigu.

9.2.1 Préemption forte ou faible

En ARGOS, la structure de préemption définie par le raffinement correspond à la préemption faible d'ESTEREL : prendre une transition issue d'un état X raffiné par P détruit P, mais il peut encore exécuter une de ses propres réactions dans la réaction globale qui le détruit (ses "dernières volontés" sont prises en compte). Cette caractéristique est nécessaire à l'expression des "suicides", dans lesquels le processus raffinant P demande à être détruit, en exécutant une transition qui émet un signal dédié, lequel déclenche une transition issue de l'état raffiné X.

Toutefois, de nombreux cas requièrent l'utilisation d'une structure de préemption *forte*, dans laquelle le processus ne réagit pas pendant la réaction qui le détruit. Pour un exposé des différentes notions de préemption dans les systèmes réactifs, voir [Ber93].

En ESTEREL, la préemption forte d'un processus P par un signal S est liée à la structure `do ... watching ...` . La préemption faible est construire à l'aide de la structure `trap T ... in ... exit T ... end`[1].
On écrit donc :

```
do                      trap T in
   P                       [ P; exit T ]
watching S              ||
                           [ await S; exit T ]
                        end T
```

Nous avons montré au chapitre 2, section 2.4.1, que la préemption forte peut être construite à partir de la préemption faible, en utilisant l'opération de définition d'horloge. Au chapitre 6, nous proposions même de définir pour cela une macro-notation.

Dans le langage mixte ARGOS+ESTEREL, nous proposons une syntaxe particulière pour la préemption forte, qui sera traduite directement en structure de préemption forte d'ESTEREL, sans passer par la construction à base d'horloge.

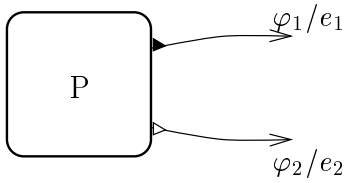


FIG. 9.1 – *Notations des préemptions faible et forte. Le triangle plein (resp. creux) dénote une préemption forte (resp. faible)*

9.2.2 Terminaison implicite

En ARGOS, il n'y a pas de notion implicite de *terminaison* d'un sous-programme. En particulier les machines de Mealy qui constituent les composants de base des programmes, si elles ont nécessairement un état *initial* distingué, n'ont pas d'état(s) final(s). D'autre part toute machine est réactive, et on ne peut utiliser la notion de *puits* pour déterminer des états de terminaison. En ESTEREL au contraire, la notion de terminaison est implicite: toute portion de programme se termine, sauf `halt` et les boucles. Les instructions élémentaires ont un début et une fin bien identifiés, et toute construction définit la terminaison du programme construit en termes des terminaisons des composants. Par exemple, la composition parallèle de n programmes se termine lorsque chacun des composants a terminé — les composants "s'attendent" les uns les autres et la terminaison de l'ensemble est simultanée à la terminaison du (ou des) dernier(s) d'entre eux. Cette synchronisation implicite des composants d'une mise en parallèle doit être explicitement construite en ARGOS, et si l'on matérialise l'attente de chacun des composants par un état, la taille de la construction obtenue est une fonction exponentielle du nombre de composants parallèles. En utilisant l'équivalent du `sustain` ESTEREL, c'est-à-dire l'émission continue d'un signal, on évite cette explosion du nombre d'états.

Exemple 9.1 : Construction de la terminaison des compositions parallèles

1. En ESTEREL v5, ces deux structures sont notées `abort ... when` et `weakabort ... when`.

On considère un système à n signaux d'entrée a_0, a_1, ..., a_{n-1}, qui émet un signal de sortie o dès qu'au moins une occurrence de chacun de ces signaux d'entrée est apparue. Pour $n = 3$ on écrit en ESTEREL :

```
Module M:
inputs a0, a1, a2;
outputs o;
   [ await a0 || await a1 || await a2  ] ;
   emit o
end.
```

En ARGOS il y a essentiellement deux solutions.

Dans la première, on écrit trois processus à une transition de la forme a_i/end_i, et on les compose en parallèle avec un synchroniseur ternaire, qui attend les trois signaux "end" et émet o. La figure 9.2 donne une version simplifiée de ce synchroniseur, dans le cas où les terminaisons des différents processus ne sont jamais simultanées (pour prendre en compte d'éventuelles terminaisons simultanées, il faut ajouter tous les "raccourcis" de deux ou trois transitions).

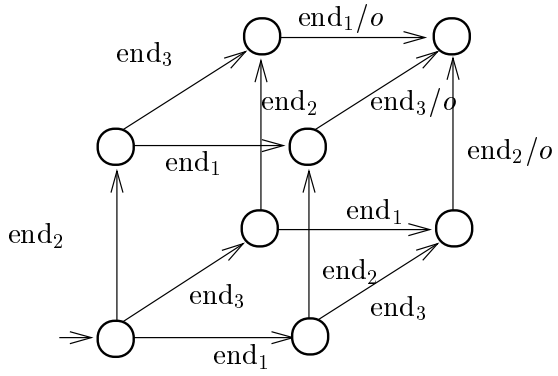


FIG. 9.2 – *Synchroniseur ternaire*

La deuxième solution évite l'explosion des états. Les processus à composer sont donnés par :

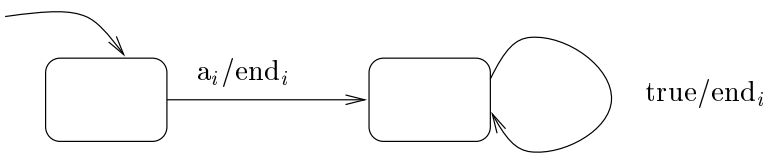


FIG. 9.3 – *Construction de la terminaison par émission continue d'un signal*

Et le synchroniseur ternaire observe simplement l'occurrence simultanée des signaux "end_1", "end_2" et "end_3". Il a donc une unique transition $end_1 \wedge end_2 \wedge end_3/o$. □

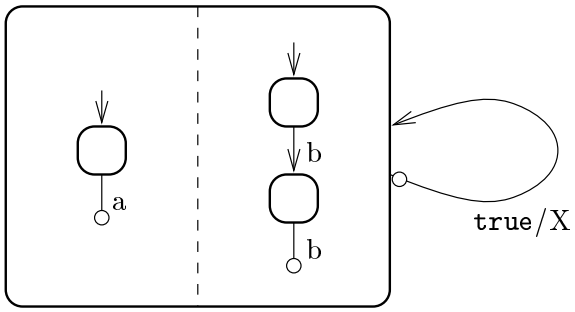


FIG. 9.4 – *Syntaxe pour la terminaison : les transitions de terminaison du processus raffinant ne conduisent pas à un véritable état : on les dessine terminées par un petit cercle ; la récupération de la terminaison par le processus raffiné est dénotée par une transition dont l'origine porte le même petit cercle.*

La deuxième construction illustrée par l'exemple ci-dessus est peu coûteuse, mais elle n'est pas modulaire : si l'on ajoute un composant numéro 4 qui se comporte comme les trois premiers, le synchroniseur doit être modifié pour réagir à $\text{end}_1 \wedge \text{end}_2 \wedge \text{end}_3 \wedge \text{end}_4$. Il est donc intéressant d'introduire la terminaison comme notion implicite.

Pour introduire la terminaison implicite dans ARGOS, on doit permettre de dénoter la terminaison des processus élémentaires. On doit également prévoir de dénoter des transitions particulières dans les automates contrôleurs de raffinements, déclenchables sur terminaison du processus raffinant. Ces transitions ne peuvent être que des préemptions faibles (le processus raffinant doit réagir, précisément pour signaler sa terminaison) et elles ont une condition d'entrée réduite à `true` (On peut aussi autoriser plusieurs transitions de récupération de la terminaison, à condition que leurs conditions soient deux à deux exclusives et que la disjonction de leurs conditions donne `true`). La figure 9.4 donne un programme qui émet X chaque fois qu'il a reçu une entrée a et deux entrées b.

Charles André propose pour les SyncCharts (voir paragraphe 9.5 ci-dessous pour plus de détails) une notion d'état "*accepteur*" plutôt que terminal. C'est une extension très intéressante qui permet de décrire des cas où la terminaison est définie de manière *distribuée*. Dans l'extension proposée ci-dessus, comme en ESTEREL, chaque processus élémentaire séquentiel a un état terminal parfaitement défini, et la terminaison d'une composition parallèle ne peut être définie que comme la terminaison de tous les processus. Ch. André donne l'exemple d'un compteur modulo 8 où la valeur 5 constitue l'état terminal. Si ce compteur est décrit par un unique automate (ou, en ESTEREL, un unique processus séquentiel), il est fort simple de définir la terminaison comme désiré. Si l'on code le compteur par la composition parallèle de trois compteurs 1 bit (c'est l'exemple 2.1 donné au chapitre 2), la situation terminale est décrite par l'état global One/Zero/One. Pourtant aucun des trois états de base ne peut être déclaré terminal dans sa composante.

En ESTEREL ou en ARGOS étendu comme ci-dessus, la terminaison implicite ne permet pas d'exprimer cette terminaison distribuée. On la code toutefois facilement par le mécanisme proposé dans l'exemple ci-dessus, où chaque état accepteur émet continuellement le signal de terminaison de sa composante.

9.2.3 Priorités de transitions

Les machines de Mealy qui constituent les composants de base des programmes ARGOS doivent être déterministes, et cela exige que les conditions booléennes des transitions issues d'un même état soient deux à deux exclusives. Nous avons vu au chapitre 6 comment relâcher quelque peu cette contrainte en proposant des conventions raisonnables d'écriture abrégée des conditions booléennes. Toutefois, dans le cas général, cette exigence de déterminisme peut conduire à écrire des conditions complexes.

Exemple 9.2
Considérons un système qui, dans son état A, doit prendre une transition t_1 lorsque le signal a_1 survient, sinon une transition t_2 lorsque a_2 survient, sinon... etc. La transition t_1 a pour condition a_1, la transition t_2 a pour condition $\neg a_1 \wedge a_2$, puis $\neg a_1 \wedge \neg a_2 \wedge a_3$, etc. □

Dans de tels cas, il pourrait être plus simple de permettre une notation syntaxique de l'ordre de priorité entre transitions, au lieu de s'en remettre à l'expression des conditions. La condition d'une transition serait alors implicitement renforcée par la négation des conditions de toutes les transitions plus prioritaires.

Dans un langage séquentiel, la priorité serait exprimée par l'ordre des conditions (la plupart des constructions de type CASE ont une interprétation séquentielle : la structure `switch` de C, la fonction `cond` de LISP et de ses dérivés, ...). Dans un langage graphique dont la syntaxe est à deux dimensions, définir un ordre sur les transitions par des critères de position (par exemple décider que les transitions issues d'un état sont ordonnées en fonction de la position de leur point de départ sur l'état, dans le sens trigonométrique) peut imposer des contraintes de placement assez pénibles. Nous proposons donc de numéroter les transitions.

9.2.4 Activation sur initialisation

En ARGOS, tout programme ou sous-programme a un *état* initial, non une *transition* initiale, comme en ESTEREL. D'un point de vue global, cette contrainte empêche de décrire un système qui, avant d'accepter des entrées, réalise quelques actions et émet quelques signaux destinés à installer une configuration initiale de l'environnement. Cet inconvénient est toutefois contournable en décrivant des systèmes munis d'un état initial fictif, et dont la première transition est en quelque sorte spontanée (sa condition est TRUE, et la transition est donc activable à chaque passage dans la boucle d'exécution décrite en introduction). L'activation de cette première transition réalise les actions désirées.

D'un point de vue local, c'est-à-dire concernant les sous-programmes, cela explique pourquoi les processus raffinants ne réagissent pas dans la réaction qui les lance, alors qu'ils réagissent éventuellement dans la réaction qui les détruit. Cette dissymétrie de comportements a quelques inconvénients illustrés par l'exemple suivant.

Exemple 9.3 : Réaction initiale et choix de l'état initial
Considérons un système à deux modes Running et Stopped. Lorsqu'il reçoit une question q, il répond par y dans l'état Running et par n dans l'état Stopped. Le mode peut changer sur occurrence d'un signal change. Dans un système plus complexe, on peut avoir besoin de plusieurs instances de ce système simple, certaines dont l'état initial est Running, d'autres dont l'état initial est Stopped. Il est bien sûr possible de décrire deux systèmes de ce type, en changeant l'état initial. Mais on pourrait se contenter d'une version *paramétrée*, en quelque sorte dynamiquement, par

une réaction initiale capable de choisir l'état initial. La figure 9.5 donne un tel système (en cas d'occurrence simultanée de la question q et du signal de changement de mode change, le système émet y et n. On pourrait bien sûr éviter ce comportement par l'expression de priorité entre transitions).

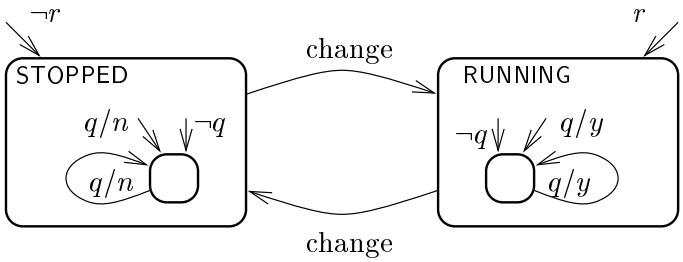


FIG. 9.5 – *Réaction initiale et choix de l'état initial*

□

La réaction initiale des sous-programmes permet donc, en particulier, le choix de l'état initial, et accroît la réutilisabilité des composants. Cette extension offre la puissance des transitions inter-niveaux entrantes des Statecharts, tout en préservant une structure propre des programmes.

On trouvera dans [JM94a, MV92a] la description de l'interface boutons d'une montre plus complexe qu'à l'ordinaire — les boutons peuvent avoir des significations différentes selon la durée d'appui, et l'on y trouve une notion analogue au double click souris — utilisant une structure de raffinement avec initialisation. Cette opération est d'ailleurs implantée dans le compilateur ARGOS actuel, mais souffre d'une étude imparfaite des problèmes de réincarnation posés par son utilisation générale. Lorsqu'on exécute une transition qui est une boucle sur un état raffiné, le processus interne existe simultanément en deux exemplaires: une instance en train de mourir, qui a droit à une ultime réaction, et une instance en cours de création, qui a droit à une réaction initiale. Ces deux instances sont décrites par la même portion de programme, mais la sémantique doit décrire comment cette portion de programme est dupliquée en deux objets aux évolutions potentiellement distinctes. La difficulté majeure provient des signaux locaux à ce processus "réincarné".

Nous proposons ici d'introduire les réactions initiales sous forme de transitions initiales dans les automates. Syntaxiquement, un automate peut avoir plusieurs transitions étiquetées et matérialisées par des flèches sans origine, portant des conditions deux à deux exclusives (ou ordonnées selon un ordre de priorité), et couvrant le domaine des entrées ; autrement dit la disjonction des conditions de ces transitions doit donner `true`. La traduction structurelle en ESTEREL permet de ne pas se préoccuper des problèmes de réincarnation, puisqu'ils sont résolus dans les compilateurs récents [Ber91] (en ESTEREL la réincarnation provient des déclarations de signaux dans une structure `loop`).

9.2.5 Etats transitoires

9.2.5.1 Séquence et états transitoires

La dernière construction d'ESTEREL qui nous intéresse, parce qu'elle est difficile à construire en ARGOS, est la *séquence* dans l'instant. L'introduction d'un mécanisme de détection de la terminaison permet d'enchaîner plusieurs processus séquentiellement, mais la séquence dans l'instant n'existe en ARGOS que sous une forme très particulière.

Pour un programme ARGOS réduit à un automate, toute réaction est constituée d'exactement *une* transition. Il y a donc identification parfaite entre les états *syntaxiques* (ceux de l'automate) et les états *sémantiques*, c'est-à-dire les états stables du système, à l'exécution. Il semble que ce soit une propriété importante lorsque l'on est habitué à concevoir des systèmes sous forme d'automates.

Les terminaisons intempestives sont codées par un mécanisme de suicide. Le processus raffinant exécute une transition t' et émet un signal α qui déclenche une transition t issue de l'état raffiné. Cette transition le tue, donc. On fait ainsi apparaître des états *transitoires*. En effet, l'état but de la transition t' ne correspond à aucun état stable de l'exécution. Dans la version compilée, il n'apparaît plus, d'ailleurs. (Si l'on généralise le raffinement pour autoriser une réaction initiale du processus lorsqu'il est lancé, son état initial devient également transitoire).

Le dialogue instantané (voir l'exemple 3.1 au chapitre 3) demande une séquence dans l'instant, et s'écrit en ESTEREL :

```
[ present C then emit Q ; present Y then ... ]
||
[ ... present Q then emit Y ... ]
```

En ARGOS on doit exprimer cette séquence par un raffinement. L'état transitoire du processus raffinant correspond à l'"état" du programme ESTEREL au niveau du point-virgule.

Toutefois un codage systématique des séquences d'ESTEREL par des raffinements ARGOS produit rapidement des niveaux d'imbrication prohibitifs. D'où l'idée de généraliser la notion d'état transitoire. Le dialogue instantané s'écrirait en ARGOS comme illustré figure 9.6. L'état en pointillé est transitoire : une transition qui y mène peut (ou doit ?) être exécutée simultanément avec une transition qui en sort. La transition d'étiquette C/Q s'enchaîne donc naturellement avec la transition Y. Les états transitoires ne doivent pas non plus constituer des puits, et il faut donc prévoir explicitement la transition par $\neg Y$; de plus cette transition n'est pas une boucle, au contraire de la transition $\neg C$, qui peut être omise.

Il est possible de transformer une machine à états transitoires en machine ordinaire, sans état transitoire, au risque de provoquer une explosion du nombre des transitions. En effet, il suffit de supprimer les états transtoires, et d'écrire explicitement tous les enchaînements possibles de transitions, entre deux états non transitoires. Il faut pour cela que la machine soit dépourvue de cycles d'états transitoires. On retrouve la condition de correction des structures de boucles d'ESTEREL : les boucles en temps nul sont rejetées.

Les états transitoires peuvent ainsi être utilisés pour coder les transitions "branchues" des Statecharts. Ces transitions représentent des arbres de décision sur les signaux. La figure 9.7 en donne un exemple. l'utilisation de branches **(a)** permet une notation concise des conditions de **(b)**, dans le style des BDD.

La traduction en ESTEREL utilise le mécanisme de traitement immédiat des signaux.

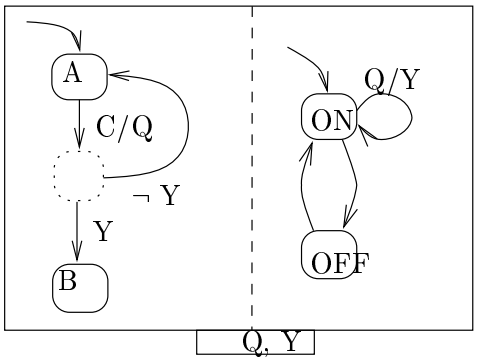


FIG. 9.6 – *Expression du dialogue instantané grâce à un état transitoire*

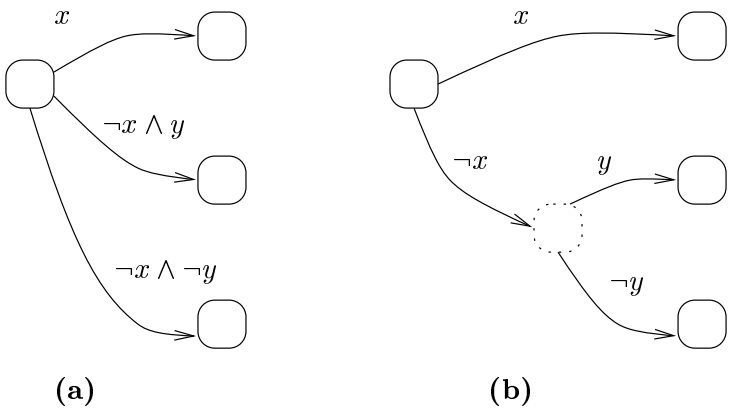


FIG. 9.7 – *Utilisation des états transitoires pour le codage des transitions branchues*

9.2.5.2 Raffinement des états transitoires

Les difficultés surviennent lorsque l'on tente d'utiliser de telles machines à états transitoires comme composants de programmes ARGOS. Peut-on raffiner un état transitoire? Si oui, doit-on obligatoirement le raffiner par un processus qui se termine toujours dans l'instant?

Nous n'avons pas poursuivi la réflexion plus loin. Il nous a semblé que le raffinement des états transitoires était un cas typique de construction intuitivement non justifiable. Restreindre l'utilisation des états transitoires aux composants de base non raffinés est une contrainte suffisante. Il est peut-être possible de la relâcher quelque peu, mais il faudrait pour cela étudier de nombreux exemples.

9.3 Schémas de traduction structurelle en ESTEREL

La compréhension détaillée des schémas de traduction présentés ici suppose une connaissance assez poussée du langage ESTEREL. Une introduction assez détaillée pour permettre au lecteur complètement novice de comprendre ces schémas ne peut, à l'évidence, être incluse dans ce document.

Toutefois nous tentons d'illustrer l'idée générale de ces schémas de traduction, qui est de fournir un cadre assez souple de traduction des machines de Mealy, tel que la prise en compte de l'une ou l'autre des extensions proposées ci-dessus soit réalisable par des modifications ponctuelles. Nous essaierons, autant que possible, de donner l'intuition des modifications introduites.

Pour un processus ARGOS P, notons $[[P]]$ sa traduction en ESTEREL. La traduction est telle que, pour tout opérateur n-aire op d'ARGOS, $[[\mathsf{op}(P_1,\ldots,P_n)]] = [[\mathsf{op}]]([[P_1]],...,[[P_n]])$.

Cette propriété est essentielle. C'est sur elle que repose la justification d'un mélange ARGOS+ESTEREL au niveau source: on pourra par exemple écrire un programme ESTEREL et l'utiliser pour raffiner un état d'automate ARGOS.

Nous esquissons ci-dessous le schéma de traduction qui permet de traduire simplement toutes les constructions d'ARGOS en ESTEREL, de manière structurelle, et se prête bien à l'introduction des extensions proposées ci-dessus.

9.3.1 Composants de base

Considérons une machine de Mealy $M = (S, \sigma_0, I, O, T)$. La traduction utilise un ensemble de signaux auxiliaires $\{\mathtt{go}_\sigma \mid \sigma \in S\}$, un pour chaque état σ de la machine. Le signal $\mathtt{go}_\sigma$ sera émis chaque fois qu'une transition qui active l'état σ est exécutée.

$[[M]]$ est un module ESTEREL de la forme:

```
module M:
input { s | s ∈ I − 0};
output { s | s ∈ O − I};
inputoutput { s | s ∈ I ∩ O};
signal { go_σ | σ ∈ S} in
      emit go_σ0 ; ||_{σ∈S} P_σ
end.
```

Le corps du module commence par l'émission du signal qui active l'état initial σ_0 de S. On trouve ensuite la composition parallèle des processus $\mathtt{P}_\sigma$ pour $\sigma \in S$, qui décrivent les états de la

machine. Chaque $\texttt{P}_\sigma$ est de la forme:

```
loop
    await immediate go_σ ;
    trap OUT_σ in
          halt
    ||
          ||_{t=(σ,−,−,−)∈T} Q_t
    end
end
```

L'instruction `halt` est ici inutile, mais elle sera remplacée par un programme, lorsque l'automate sera raffiné.

Finalement le codage Q_t d'une transition $t = (\sigma, \varphi, e, \sigma') \in T$ est de la forme:
`await` φ`; emit e; emit go`$_{\sigma'}$`; exit OUT`$_\sigma$.

Le programme attend l'événement décrit par φ, puis émet les signaux de l'ensemble de sortie e, et le signal $\texttt{go}_{\sigma'}$ qui active l'état but de la transition, puis sort de la structure "`trap OUT`$_\sigma$...", ce qui désactive l'état source σ. Notons que, si la transition est une boucle (c'est-à-dire $\sigma = \sigma'$), la portion de programme qui décrit l'état σ est simultanément réactivée, conformément à la sémantique d'ARGOS.

La notation `await` φ doit être comprise comme une abréviation de:

- "`halt`" si $\varphi =$ FALSE
- "`await tick`" si $\varphi =$ TRUE
- "`await` $[\![\varphi]\!]$" sinon, avec: $[\![\texttt{x}]\!] = \texttt{x}$, $[\![-\varphi]\!] =$ `not` $[\![\varphi]\!]$, $[\![\varphi_1 + \varphi_2]\!] = [\![\varphi_1]\!]$ `or` $[\![\varphi_2]\!]$ and $[\![\varphi_1.\varphi_2]\!] = [\![\varphi_1]\!]$ `and` $[\![\varphi_2]\!]$.

Ce choix de codage des transitions issues d'un même état appelle deux remarques.

Tout d'abord, on notera que les portions de programme codant les différentes transitions sont composées *en parallèle*, ce qui permet de ne pas imposer de priorité statique entre elles (ce qui serait le cas si on les traduisait par des structures de chien de garde nécessairement emboîtées). Cet aspect de la traduction est essentiel pour respecter la sémantique d'ARGOS. En effet, il y a un côté *declaratif* plutôt qu'*impératif* dans l'interprétation de l'ensemble de transitions issues d'un même état: aucun ordre n'est imposé entre elles. Cette réflexion motivait d'ailleurs le titre "An experience in compiling a mixed declarative/imperative language for reactive systems" de l'article [MV92b] dans lequel nous décrivions l'utilisation des BDD dans le compilateur ARGOS.

Le deuxième point important concerne le codage des transitions, individuellement. On traduit la structure φ/e par une structure séquentielle d'ESTEREL: `await` φ `; emit e`, qui introduit une dépendance entre l'émission de e et la présence ou l'absence des signaux qui déterminent l'événement φ. En ESTEREL ces dépendances entre signaux (déduites des structures séquentielles, conditionnelles, ou des structures de contrôle temporelles) forment la base de la définition de la notion de *causalité* des programmes. Ces dépendances conduisent parfois à rejeter, selon des critères de causalité, un programme ESTEREL obtenu par traduction d'un programme ARGOS, alors même que ces critères n'ont pas de sens en termes du programme ARGOS d'origine. Le paragraphe 9.6 ci-dessous précise ces idées.

9.3.2 Traduction des constructions d'ARGOS

La traduction des constructions d'ARGOS est directe : la mise en parallèle est traduite en mise en parallèle ESTEREL ; l'encapsulation en structure de déclaration `signal ... in ... end` ; la définition d'horloge en structure `suspend ... when ...`. Le raffinement d'une machine M par un ensemble de processus P_σ est obtenu en remplaçant, dans la traduction ci-dessus de la machine M, l'instruction `halt` de P_σ par `run` R_σ, où R_σ est un module ESTEREL quelconque, écrit directement en ESTEREL ou issu de la traduction d'un autre programme ARGOS.

9.3.3 Traduction des extensions proposées

9.3.3.1 Préemptions faible et forte

La traduction en ESTEREL vise à préserver une certaine symétrie parmi les différents types de transitions : aucune priorité n'est imposée a priori entre les transitions qui représentent des préemptions faibles et celles qui représentent des préemptions fortes. Nous montrons au paragraphe 9.3.3.3 que cela permet d'éviter l'introduction de "faux" problèmes de causalité.

On ne change que le codage $\mathtt{P}_\sigma$ d'un état σ. Dans la suite, nous noterons $\mathtt{R}_\sigma$ le codage du processus raffinant l'état σ, s'il y en a un, et l'instruction `halt` sinon.

```
loop
    await immediate go_σ ;
    signal weak, strong in
           trap OUT_σ in
               do
                 run R_σ
               ||
                 await weak; exit OUT_σ
               watching strong timeout exit OUT_σ end
           ||
                    ||          Q_t
               t=(σ,−,−,−)∈T
           end trap
    end signal
end loop
```

et les codages de transitions Q_t, $t = (\sigma, \varphi, e, \sigma') \in T$, sont de la forme :

```
await φ; emit e; emit go_σ' ;
```

`emit` $\begin{cases} \mathtt{weak} & \text{si la transition est faible} \\ \mathtt{strong} & \text{si la transition est forte} \end{cases}$

9.3.3.2 Terminaison

Ici on ne change que la structure générale du module qui code une machine raffinée. Le corps du module précédemment défini est englobé dans une construction `trap` :

```
module M :
input ...
```

```
output ...
inputoutput ...
signal... in
      trap TERM in
           emit go_{σ0} ; ||_{σ∈S} P_σ
      end trap
end.
```

Une transition de terminaison de l'un des processus raffinants doit simplement être traduite en ESTEREL par une séquence d'instructions qui se termine par `exit TERM`, au lieu de l'émission du signal `go` de son état but.

La récupération de la terminaison par l'automate contrôleur d'un raffinement est réalisée en deux points: on introduit un signal $\texttt{term}_\sigma$ local au programme d'un état σ, et on fait suivre l'instruction $\texttt{run R}_\sigma$ de lancement du processus qui raffine σ de l'instruction $\texttt{emit term}_\sigma$. Dans le programme de traitement des transitions, les transitions spéciales de récupération de la terminaison doivent être conditionnées par la présence de $\texttt{term}_\sigma$, ce qui donne donc: $\texttt{await } \varphi \texttt{ and term}_\sigma \texttt{ ; emit e; emit go}_{\sigma'} \texttt{; exit OUT}_\sigma$. comme codage d'une transition de terminaison $t = (\sigma, \varphi, e, \sigma') \in T$.

9.3.3.3 Priorités de transitions

La traduction en ESTEREL des niveaux de priorité est directe, puisqu'il existe dans ce langage une structure de type CASE à interprétation séquentielle. Il suffit de considérer que les portions de programme qui codent les transitions issues d'un même état ne sont plus en parallèle mais structurées par un `case`:

```
await
     case φ1 do emit e1; emit ...
     ⋮
     case φn do emit en; emit ...
end
```

Cette traduction est compatible avec l'utilisation de transitions de terminaison, et avec l'expression de préemptions fortes ou faibles. Il faut toutefois noter que donner une plus grande priorité à une transition de préemption faible, qu'à une transition de préemption forte, a de fortes chances de conduire à des problèmes de causalité assez subtils.

9.3.3.4 Réaction initiale

Dans le schéma de traduction d'une machine de Mealy, il suffit de remplacer l'instruction $\texttt{emit go}_{\sigma_0}$ qui permettait de définir l'état initial, par un traitement de transition initiale. Si les conditions des transitions initiales sont exclusives, on obtient par exemple:

```
[
        present φ1 then
                emit e1;
                emit go_{σ'1}
```

```
        end
||      ...
||      present φ_n then
                emit e_n ;
                emit go_σ'_n
        end
]
```

9.4 Preuve de conformité

La traduction d'ARGOS étendu en ESTEREL en donne une sémantique formelle, puisqu'ESTEREL dispose d'une sémantique formelle.

Pour le langage ARGOS de base, il serait bon de comparer la sémantique usuelle par compilation en machines de Mealy, à la sémantique définie par traduction en ESTEREL. C'est très certainement une preuve par induction structurelle assez lourde, mais sans véritable difficulté autre que la notion de correction des programmes. Le paragraphe 9.6 ci-dessous expose les problèmes.

9.5 Travaux de même nature

9.5.1 Les SyncCharts

Charles André et son équipe (Université de Nice) travaillent à la définition du langage SyncCharts [And96, AG95]. C'est un langage graphique à base d'automates hiérarchiques comme en ARGOS, qui hérite de certaines constructions ESTEREL — modes de terminaison, réaction initiale des processus raffinants, traitement immédiat des signaux — et de constructions du Grafcet — association de l'émission d'un signal à un état, inhibition de processus par un signal, etc. L'ensemble est traduit en ESTEREL.

9.5.2 Synopsys

Joe Buck chez Synopsys travaille sur une extension d'ARGOS dans laquelle on retrouve essentiellement les différents types de transitions associés aux modes de terminaison ESTEREL (terminaison normale, préemptions faible et forte) et la réaction du processus raffinant lors de son activation. L'ensemble est traduit en ESTEREL.

9.6 Comparaison des notions de causalité en ARGOS et ESTEREL

9.6.1 Un point de vue sur la causalité

Nous avons vu au chapitre 2 comment interpréter l'encapsulation ARGOS en termes de systèmes d'équations booléennes à résoudre pour déterminer le statut des signaux locaux. Au chapitre 5 nous montrons comment traduire l'ensemble des constructions d'ARGOS en équations booléennes. Il existe pour ESTEREL "pur" (c'est-à-dire réduit aux booléens) une traduction complète en systèmes d'équations booléennes [Ber91].

L'exigence de déterminisme et de réactivité des programmes décrits s'exprime, comme pour ARGOS, par le fait que ces systèmes d'équations doivent avoir une solution unique. Le statut des

signaux locaux — sur lesquels repose la synchronisation entre composants par diffusion synchrone — doit être parfaitement déterminé, pour tout état, pour toute configuration des signaux d'entrée.

L'exigence de *causalité* des programmes — le terme a été introduit pour ESTEREL — va plus loin. Une manière de voir les choses est de dire qu'un programme est non causal si le système d'équations sous-jacent, qui définit le statut des signaux locaux, admet des solutions inexplicables en termes des structures syntaxiques du langage. On voit là que la notion de causalité ne peut qu'être dépendante du langage.

9.6.2 Comment définir une notion de causalité pour ARGOS? une conjecture...

Dans l'état actuel de la sémantique proposée pour ARGOS, il n'y a pas de notion de *causalité*. La sémantique se contente de définir une notion de programme incorrect vis-à-vis du déterminisme ou de la réactivité. L'implantation qui utilise l'outil BAC (cf. chapitre 5) permet d'affaiblir quelque peu les critères sans remettre en cause l'exigence de déterminisme et réactivité. On accepte toujours tout programme qui définit complètement l'état but et les signaux émis, dans tout état accessible, pour tout événement d'entrée.

Le besoin éventuel d'une notion de causalité (c'est-à-dire de critères plus *forts* que la simple exigence de déterminisme et réactivité) apparaît lorsque l'on considère des programmes comme le dialogue instantané décrit au chapitre 3, page 47. Dans la version **(b)**, il est impossible d'expliquer la transition globale par un enchaînement de transitions élémentaires, ce qui peut paraître curieux. Cette remarque pourrait conduire à adopter la sémantique décrite pour les "réactions en chaîne" (toujours au chapitre 3) dans laquelle, outre l'existence d'une solution unique au système d'équations qui définit le statut des signaux locaux, on exige l'existence d'un ordre entre les transitions des composants de base. Le dialogue instantané ne peut alors s'écrire qu'avec un raffinement. ce choix aurait pour avantage de fournir une explication très simple de la non causalité d'un programme en termes de structures syntaxiques d'ARGOS.

D'autre part, si l'on traduit ARGOS en ESTEREL, on hérite nécessairement de la notion de causalité d'ESTEREL. Lorsqu'une traduction est rejetée par le compilateur ESTEREL, il faudrait savoir le justifier en termes des structures syntaxiques d'ARGOS. Cela risque d'être compliqué, ou artificiel.

La causalité d'ESTEREL repose sur une interprétation en logique constructive des systèmes d'équations booléennes qui définissent le statut des signaux locaux. Si l'on pouvait montrer que cette notion coïncide avec la sémantique par enchaînement de transitions élémentaires mentionnée ci-dessus, cela donnerait une notion solide de causalité pour ARGOS, et autoriserait la compilation par traduction en ESTEREL. Rappelons que la traduction d'ARGOS en IC (format intermédiaire du compilateur ESTEREL antérieur à l'analyse de causalité) avait été abandonnée en 1990, essentiellement parce que le compilateur V3 exhibait des erreurs de causalité incompréhensibles au niveau du source ARGOS.

Observons un exemple, figure 9.8.

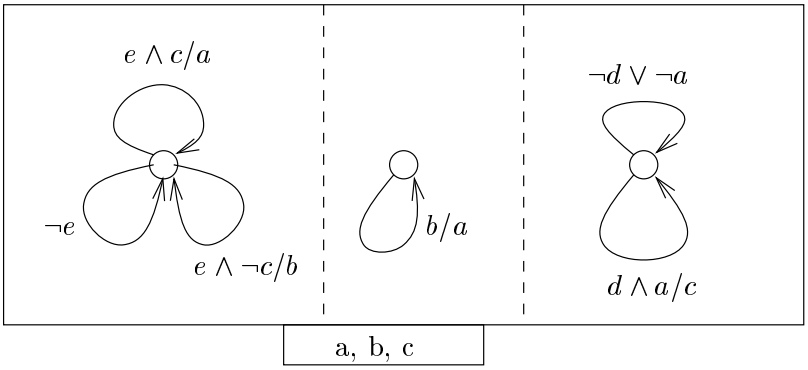


FIG. 9.8 – *Dépendances entre signaux résolues par calcul booléen classique*

Pour l'unique état global, les équations des signaux locaux sont :

$$a = e \wedge c \vee b$$
$$b = e \wedge \neg c$$
$$c = a \wedge d$$

qu'une simplification booléenne élémentaire réduit à :

$$a = e$$
$$c = e \wedge d$$
$$b = e \wedge \neg d$$

où les dépendances ont disparu. Le programme ARGOS est donc correct.

Or, quelle que soit la valuation des entrées e et d, la transition globale déduite du statut des signaux locaux défini par $a = e, c = e \wedge d, b = e \wedge \neg d$ n'est pas explicable par un enchaînement de transitions élémentaires.

D'autre part la simplification effectuée n'est pas autorisée en logique constructive, puisqu'elle nécessite l'utilisation de l'axiome $a \vee \neg a = \text{TRUE}$.

S'agit-il donc de la même notion ? Cette question est ouverte. Il semble assez évident que, s'il existe pour toute configuration des entrées un enchaînement de transitions élémentaires qui détermine le statut des signaux locaux, alors le système d'équations booléennes a une solution en logique constructive, qui n'est pas réduite au n-uplet indéfini.

Mais peut-être existe-t-il des situations complexes dans lequel le système d'équations a une solution constructive, bien que l'on échoue toujours à l'expliquer par un enchaînement de transitions élémentaires.

9.6.3 Pourquoi a/b n'est pas toujours une commande gardée

Quelle que soit la réponse à la question ci-dessus, on peut aussi s'interroger sur une notion de causalité ARGOS qui impose que les réactions globales soient toujours explicables comme enchaînement de transitions élémentaires, ce qui correspond à une interprétation très impérative de la notation a/b.

Ce paragraphe illustre simplement une classe d'exemples pour lesquelles la notation a/b n'est pas interprétée comme une commande gardée. Dans ce cas l'exigence d'enchaînement de transitions peut conduire à rejeter des programmes qu'on aimerait conserver. De même la traduction

en ESTEREL `present a then emit b`, qui fait dépendre l'émission de b de la présence de a, conduira à des erreurs de causalité non interprétables en ARGOS.

L'interprétation de a/b comme une commande gardée signifie que cette notation se lit : *lorsque je suis sûr(e) de la présence de a, j'émet b*. C'est un point de vue très impératif.

Une autre interprétation possible est : *la présence de a est une condition suffisante à l'émission de b*. Peut-être faudrait-il d'ailleurs adopter une notation du genre $b \leftarrow a$, pour mieux refléter l'aspect déclaratif de cette affirmation. Rien ne dit que la présence de a soit nécessaire à la présence de b.

Exemple 9.4 : Conception dirigée par les entrées ou par les sorties
Considérons un système qui, dans un état A, doit émettre e lorsque a et b sont présents simultanément, et $\{e, f\}$ lorsque a est présent et b absent. Avec la notation suggérée ci-dessus :

$$e \leftarrow a \wedge b$$
$$e, f \leftarrow a \wedge \neg b$$

On détermine les ensembles de signaux à émettre et, pour chaque ensemble, on indique une condition suffisante. Un tel ensemble de définitions peut être augmenté de nouvelles conditions suffisantes à l'émission de l'ensemble $\{e, f\}$ par exemple.

Une vision impérative de cette situation donnera toujours une structure de tests sur la présence de a et b, provoquant l'émission de e, f ou $\{e, f\}$. Une longue habitude de factorisation des tests, héritée de la programmation séquentielle, conduira à minimiser la duplication de code en écrivant un arbre de décision. En ESTEREL on obtient quelque chose du genre :

```
present a then
    emit e ;
    present b else emit f end
end
```

où il apparaît très clairement que l'émission de e ne dépend pas du statut de b.

Pour décrire une telle situation avec un automate (où l'on ne dispose pas a priori des transitions branchues proposées plus haut), on dessinera plutôt deux transitions issues du même état A, d'étiquettes $a \wedge b/e$ et $a \wedge \neg b/e, f$. L'information selon laquelle l'émission de e ne dépend pas du statut de b est en quelque sorte *syntaxiquement distribuée*, mais elle est toujours là. Cette petite différence de syntaxe doit-elle conduire à une différence majeure de sémantique? □

Dans l'exemple précédent, on pourrait imaginer des méthodes de codage, ou des extensions de la syntaxe des automates (comme les transitions branchues) qui éviteraient d'écrire de fausses dépendances entre signaux. Cela peut être fait localement, si les transitions $a \wedge b/e$ et $a \wedge \neg b/e, f$ appartiennent au même automate. Toutefois, si le même genre de situation se présente dans une composition parallèle, ce n'est plus possible. On retombe sur l'exemple de la figure 9.8 ci-dessus.

Troisième partie

Extensions temporelle et hybride

Chapitre 10

Argos temporisé

Sources : *L'extension temporisée d'*ARGOS *est présentée dans [JMO93]. La méthode est exposée sur un exemple dans [JM95]. D'autre part le compilateur* ARGOS *étendu aux automates temporisés, couplé à l'outil* KRONOS*, a servi de support à l'activité de Muriel Jourdan en stage post-doctoral. Il s'agissait d'appliquer la méthode proposée dans un environnement de programmation de robots mobiles. Ce travail est rapporté dans [Jou96, SKEJ95].*

10.1 Motivations

Les motivations de ce travail sont de deux ordres, selon que l'on adopte le point de vue des concepteurs de langages synchrones ou celui des utilisateurs d'automates temporisés.

10.1.1 Prise en compte du temps dans les langage synchrones

La notion de *temps* dont on dispose dans les langages synchrones est un temps *discret* et *multiforme*. Le temps est assimilé aux instants de présence d'un signal extérieur. Rien n'empêche, donc, de considérer différentes échelles de temps non corrélées. L'exemple standard proposé par Gérard Berry fait intervenir des échelles de temps "mètres" et "secondes" pour expliquer qu'il est plus naturel d'exiger d'une automobile qu'elle s'arrête dans un délai de 5 mètres, que dans un délai de 10 secondes.

Lorsqu'on compile un programme qui fait intervenir des délais, ou compteurs de temps, pour obtenir un automate interprété (voir définition dans le paragraphe d'introduction sur les modèles, 1.2), il y a essentiellement deux solutions.

La première solution consiste à expanser les valeurs possibles des compteurs en *états*. On risque l'explosion du nombre d'états du programme global, au moins par composition parallèle des compteurs. D'autre part, on peut avoir à considérer en parallèle des composants qui comptent le même temps avec des ordres de grandeur différents. En effet, si un composant compte des secondes, et l'autre des heures, il faut se ramener à la même unité si l'on veut pouvoir comparer les compteurs. Le compteur d'heures atteint alors, en nombre de secondes, de très grandes valeurs. Toutefois cette solution permet de prendre en compte les valeurs des compteurs dans la vérification formelle des programmes, puisque l'information correspondante existe bien dans la partie contrôle de l'automate interprété. L'expression des propriétés est quelquefois compliquée par le fait que les valeurs des compteurs sont en quelque sorte codées en base 1 : exprimer $x > y$

lorsque les valeurs des compteurs x et y sont explicitement représentées par des états n'est pas de tout repos. C'est en ce sens que nous écrivions, dans l'introduction de [JMO93], qu'il est impossible d'exprimer des propriétés *quantitatives*.

La deuxième solution consiste à coder les compteurs par des *variables* de l'automate interprété. On supprime le problème d'explosion d'états, d'une part parce que la mise en parallèle n'a pas d'effet néfaste sur les conditions et affectations qui portent sur les variables ; d'autre part parce que les compteurs d'ordres de grandeur différents se traduisent par des valeurs d'ordres de grandeur différents à l'exécution, aisément comparables. L'inconvénient majeur de cette approche est l'impossibilité de prendre en compte les aspects de délai dans l'expression de propriétés effectivement vérifiables.

La troisième solution (!) est décrite dans [Hal93]. Pour joindre les avantages de la deuxième solution ci-dessus à la possibilité d'exprimer des propriétés temporelles — avantage dont ne jouissait jusque là que la première des deux solutions — il "suffit" d'appliquer aux automates interprétés issus par exemple du compilateur ESTEREL des techniques d'évaluation approchée, après avoir réalisé une analyse du format OC pour repérer parmi les variables au nom évocateur comme `__v__0_1`, celles qui proviennent de l'expansion des structures de contrôle temporelles. Nous retrouvons cette approche au chapitre 11, qui décrit le prototype HARGOS basé sur la connexion d'un compilateur ARGOS qui accepte des "pragmas" à l'outil POLKA de N. Halbwachs. Le seul inconvénient de la méthode réside dans la nature approchée des résultats obtenus, mais l'expérience a prouvé depuis que, dans la plupart des cas traités, l'analyse produit des résultats suffisants. Par ailleurs la même méthode (et le même outil d'analyse) peut être utilisée pour découvrir la valeur d'un délai permettant d'assurer une propriété donnée, en considérant certaines des valeurs de délai, non plus comme des constantes, mais comme des variables.

Le travail présenté dans [JMO93] explore une voie légèrement différente. En acceptant de perdre la richesse d'expression du temps multiforme, propre aux langages synchrones, on étend ARGOS avec une construction de délai, et on compile les programmes en automates temporisés (*Timed graphs* [AD90]), modèle sur lequel des propriétés quantitatives exprimées en logique temporelle temps-réel sont vérifiables. Pratiquement, on connecte le compilateur ARGOS à l'outil KRONOS développé à Vérimag, via son format d'entrée.

10.1.2 ARGOS temporisé comme langage de description des automates temporisés

Du point de vue des utilisateurs d'automates temporisés, la définition d'ARGOS temporisé fournit un *langage*. En effet les automates temporisés, comme tout modèle à base de systèmes de transitions, ne peuvent être utilisés directement pour la description de systèmes un tant soit peu complexes. Toutefois ARGOS temporisé n'a pas la puissance d'expression maximale. La construction de délai ne produit que des automates très particuliers. La question se pose alors de déterminer la classe des automates temporisés accessibles. Il est assez facile de caractériser cette classe de manière syntaxique, mais pour que cela offre de l'intérêt il faudrait savoir comparer la caractérisation syntaxique et les équivalences comportementales d'automates temporisés.

Muriel Jourdan a prouvé dans sa thèse [Jou94] que les automates temporisés obtenus par compilation d'ARGOS temporisé ont une bonne propriété : ils sont "*non zénon*" (les évolutions temporelles d'un automate temporisé qui n'a pas cette propriété supposent que le temps est borné : il croît indéfiniment, mais tend vers une borne supérieure. On se débrouille en général pour éliminer ces automates-là, pour lesquels la validité de certaines propriétés pose des problèmes d'interprétation). Ne pas avoir la puissance d'expression maximale paraît, dans ce cas, plutôt un

avantage qu'un inconvénient.

Toutefois Sergio Yovine a proposé des exemples qui montrent que l'on aimerait utiliser certains automates temporisés inaccessibles par compilation d'ARGOS temporisé. Il s'agit en général d'automates temporisés qui présentent un cas de non-déterminisme temporel.

La généralisation du travail de définition de sémantique de [JMO93], à des compositions d'automates qui portent des pragmas quelconques (voir chapitre 11), devrait fournir une solution à ce problème, au prix d'une utilisation plus difficile du langage. Toutefois, si pour des automates temporisés obtenus par compilation de la construction de délai proposée ici, l'interprétation de la synchronisation par diffusion synchrone ne posait pas de problème, cela demande à être étudié de plus près dans le cas des automates temporisés généraux, en suivant par exemple les idées de [SY96].

10.2 ARGOS temporisé et sa traduction en automates temporisés

10.2.1 Une construction temporelle

La construction temporelle que nous introduisons dans ARGOS provient d'un des tous premiers rapports sur les Statecharts, datant de 1984, qui proposait une série d'extensions à un formalisme encore tout jeune. On y trouvait donc, en particulier, la suggestion de temporiser les états. Cela consiste à associer à un état A un couple de valeurs réelles a et b, avec en général $a \leq b$. La sémantique de cette construction exprime que, une fois entré dans l'état A, le système ne peut en sortir avant qu'un délai a ne se soit écoulé, et doit en être sorti lorsque le délai b expire. Pour que le système ne se bloque pas, on prévoit une transition dite de "time out", qui force à quitter l'état A lorsque le délai b expire, si on ne l'avait pas quitté auparavant en suivant le fonctionnement normal.

Puisque les états sont en fait des macro-états, l'utilisation de cette construction sur des états raffinés permet l'expression de chiens de garde.

Les constructions temporelles proposées pour les Statecharts dans des articles plus récents sont, à mon avis, moins propres que celle-là, dont on peut déplorer la disparition.

En ARGOS, on introduit la même construction, en ne gardant que la borne supérieure. La construction à deux bornes peut être fabriquée à partir de la construction à une seule borne. On peut considérer cette nouvelle construction comme une macro-notation, et décrire formellement son expansion en ARGOS usuel. Chaque temporisation d'état a pour effet d'ajouter un composant compteur, en parallèle. On synchronise le composant temporisé avec le compteur. On y gagne en lisibilité des programmes, mais c'est une façon d'appliquer la première solution mentionnée en introduction : la taille des modèles explose.

La bonne idée consiste donc à considérer cette nouvelle construction comme faisant partie du langage, et à redéfinir la sémantique d'ARGOS en automates temporisés.

10.2.2 ARGOS temporisé

Les programmes ARGOS temporisés adoptent la même syntaxe que les programmes ARGOS simples. Les machines qui constituent les composants de base sont équipées de temporisations d'états :

$(S, s_0, I, O, T, \mathcal{T})$, où $T \in S \times \mathcal{N}(\mathcal{A}) \times 2^{\mathcal{A}} \times S$ est l'ensemble des transitions. La condition d'une transition est un élément de $\mathcal{N}(\mathcal{A}) = \mathcal{B}(\mathcal{A}) \cup \{\texttt{to}\}$, où `to` désigne une condition d'expiration de

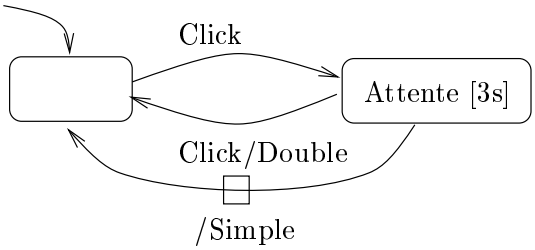


FIG. 10.1 – *Détection des doubles clicks en* ARGOS *temporisé*

délai (*timeout*). $\mathcal{T} : S \longrightarrow \mathbb{N}^* \cup \{+\infty\}$ est la fonction de temporisation des états. $\mathcal{T}(s) = +\infty$ lorsque l'état n'est pas temporisé. $\mathcal{P}_t$ désigne l'ensemble des programmes ARGOS temporisés.

La figure 10.1 donne un exemple d'utilisation des états temporisés dans un programme qui détecte les doubles clicks d'une souris. Le système reçoit en entrée un signal Click et le signal temporel. Il émet les signaux Simple ou Double, selon que le click est double — c'est-à-dire suivi d'un autre click dans un intervalle de temps inférieur à 3 unités de temps — ou simple. Ce petit programme peut servir d'interface dans un programme plus complexe qui associe des comportements différents aux clicks simple et double.

Nous avons vu au chapitre 6 comment ré-écrire un tel programme en ARGOS non temporisé, en considérant les états temporisés comme une macro-notation. Nous montrons ci-dessous comment traduire cette construction directement en automate temporisé.

10.2.3 Aperçu des automates temporisés

Un automate temporisé décrit l'évolution d'un système dans le temps grâce à un ensemble de variables réelles appelées *horloges*, qu'on peut tester et remettre à zéro. Leur valeur évolue de manière continue.

Nous utilisons les automates temporisés munis d'invariants sur les états, comme ceux définis dans [HNSY92]. Ajouter ces invariants est un moyen d'exprimer l'urgence des transitions, ce que ne permettait pas la définition initiale donnée dans [AD90] (un autre moyen d'exprimer l'urgence est décrit dans [SY96]).

La figure 10.2 est adaptée d'un exemple de [ACH+92].

La transition t de s0 à s1 initialise l'horloge x. La transition de s1 à s2 initialise l'horloge y, et peut être tirée après un temps quelconque passé dans l'état s1, pendant lequel x croît. La transition de $s2$ à $s3$ ne sera possible que si le temps écoulé depuis que t a été tirée appartient à l'intervalle $]1, 6[$. Toutefois la transition de s1 à s2, nécessairement tirée entre temps, a initialisé l'horloge y, et on ne peut être dans l'état s3 qu'à condition que $y \leq 5$. Les invariants d'états réduisent ainsi les comportements possibles. L'étiquette de la transition de s3 à s4 indique qu'elle est tirable n'importe quand dès que $y \geq 4$. L'invariant $y \leq 5$ de son état de départ la rend obligatoire quand y atteint 5.

La forme des conditions — des comparaisons à des constantes entières — et des actions d'affectation — uniquement des remises à zéro — garantit la décidabilité de la logique TCTL sur le modèle des automates temporisés. En particulier l'accessibilité d'un certain état du graphe, avec une certaine valuation des horloges, est décidable. TCTL est une version temporisée de CTL, dans laquelle on associe des bornes temporelles aux modalités comme INEV. On peut ainsi

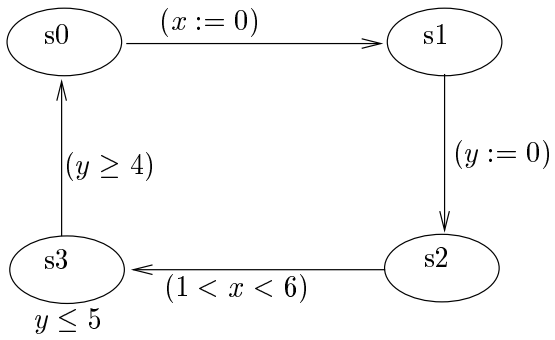


FIG. 10.2 – *Exemple d'automate temporisé*

exprimer qu'une certaine propriété sera vraie inévitablement à partir d'un état, et cela dans un délai de 5 unités de temps. L'outil KRONOS développé à Vérimag est basé sur les résultats décrits dans [HNSY92].

10.2.4 Traduction

Nous donnons la sémantique d'ARGOS temporisé en termes d'*automates temporisés avec actions*, ce qui permet de conserver les étiquettes ARGOS comme étiquettes des automates temporisés. Les automates temporisés avec actions et invariants d'états sont de la forme: $(S, s_0, I, O, T, \mathcal{F}, X)$ où X est l'ensemble des horloges, $T \subseteq S \times \mathcal{C}(X) \times \mathcal{B}(I) \cup \{\text{EMPTY}\} \times 2^O \times \mathcal{RA}(X) \times S$ est l'ensemble des *transitions* et $\mathcal{F} : S \longrightarrow \mathcal{C}(X)$ décrit les invariants associés aux états.

$\mathcal{C}(X)$ est l'ensemble des expressions booléennes construites à partir des variables de X selon la grammaire: $c ::= x < k \mid x = k$ où k est une constante entière positive ou nulle et $x \in X$ (la définition usuelle des automates temporisés autorise des conditions plus générales). $\mathcal{RA}(X) = 2^X$ est l'ensemble des actions de remise à zéro. Une remise à zéro détermine quelles variables doivent être remises à zéro lorsque la transition est exécutée. La partie action de l'étiquette en provenance d'ARGOS a la forme usuelle: c'est un élément de $\mathcal{B}(I) \times 2^O$. On ajoute l'action EMPTY pour reconnaître les transitions construites pour exprimer l'expiration des délais, qui sont en quelque sorte spontanées: elles peuvent être exécutées sans influence extérieure. Les transitions qui portent cette action sont dites *temporelles*. Noter qu'une transition temporelle peut émettre des sorties. Une transition $t = (s_i, C, m, o, r, s_f)$ sera notée $s_i \xRightarrow{C.\ m/o\ .r} s_f$.

Le principe de la traduction est simple. On associe une horloge *Var (A)* à chaque état A du programme ARGOS (lorsque l'état n'est pas temporisé, on verra apparaître des conditions de la forme $x < +\infty$ ou $x = +\infty$ que l'on remplace par `true` et `false`). Pour chaque état de chaque automate, on obtient un état d'automate temporisé avec actions, comme illustré figure 10.3.

On étend ensuite la sémantique des opérations d'ARGOS à des objets de cette forme. Nous ne donnons ici que la composition parallèle.
Soit $\mathcal{S}'(P_i) = (S_i, s_{0i}, T_i, \mathcal{F}_i, X_i)$ la traduction du programme P_i, $i \in \{1, 2\}$.
La traduction $\mathcal{S}'(P_1 \| P_2)$ est alors de la forme:

$$(S_1 \| S_2, s_{01} \| s_{02}, T', \lambda s_1 \| s_2 \bullet \mathcal{F}_1(s_1) \wedge \mathcal{F}_2(s_2), X_1 \cup X_2)$$

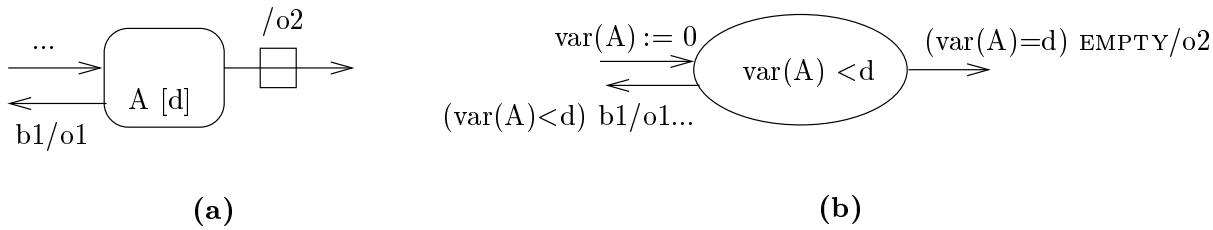


FIG. 10.3 – *Principe de la traduction*

L'ensemble des états et l'état initial sont définis comme auparavant :

$$S_1 \| S_2 = \{s_1 \| s_2, s_1 \in Q_1, s_2 \in Q_2\} \qquad s_{01} \| s_{02}$$

on fait l'union des ensembles d'horloges et l'invariant d'un état $s_1 \| s_2$ est la conjonction des invariants de s_1 et de s_2. Finalement T' est défini d'après T_1, T_2 par :

$$\frac{s_1 \xrightarrow{C_1.\, m_1/o_1\, .A_1} s_1', \quad s_2 \xrightarrow{C_2.\, m_2/o_2\, .A_2} s_2', \quad (m_1 = m_2 = \text{EMPTY}) \vee (m_1 \neq \text{EMPTY} \wedge m_2 \neq \text{EMPTY} \wedge (m_1 \wedge m_2 \neq \texttt{false}))}{s_1 \| s_2 \xrightarrow{C_1 \wedge C_2.\, m_1 \wedge m_2/o_1 \cup o_2\, .A_1 \cup A_2} s_1' \| s_2'} \quad [\text{Pt}]$$

X_1 and X_2 sont disjoints. $C_1 \wedge C_2$ ne peut être réduit à `false`, puisque les deux fonctions portent sur des ensembles de variables disjoints. Les transitions temporelles (resp. non temporelles) peuvent être exécutées simultanément. Les deux types de transitions ne peuvent être fusionnés.

10.2.5 Propriété de cohérence des sémantiques

Pour juger de la nouvelle sémantique introduite pour ARGOS temporisé, il faut trouver un moyen de la comparer à la sémantique d'ARGOS simple.

On montre que : $\mathcal{S}(\mathcal{E}xp(P_t)) \equiv \mathcal{D}(\mathcal{S}'(P_t))$ où:

- $\mathcal{E}xp$ représente l'expansion de la macro-notation de temporisation des états (cf. chapitre 6). $\mathcal{E}xp(P_t)$ est un programme ARGOS simple, qui utilise un signal "temporel" noté χ;
- Pour comparer $\mathcal{S}(\mathcal{E}xp(P_t))$ — un système de transitions avec des signaux χ, à $\mathcal{S}'(P_t)$ — un automate temporisé, où le temps est implicite — nous utilisons une sémantique en temps discret $\mathcal{D}$ des automates temporisés obtenus par traduction des programmes ARGOS. $\mathcal{D}(atg)$ est un système de transitions étiquetées où apparaît également χ.
- $\equiv$ est la bisimulation.

10.3 Implantation et exemples d'utilisation

10.3.1 Implantation

L'implantation est réalisée par modification du compilateur ARGOS existant. On produit des automates temporisés au format d'entrée de KRONOS, où le mécanisme d'étiquetage des transitions permet de conserver l'information en provenance des étiquettes de transitions ARGOS. Il est

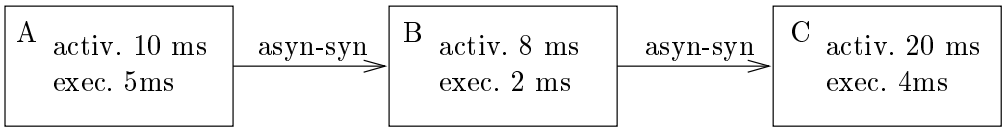


FIG. 10.4 – *Un schéma de synchronisation de tâches-modules*

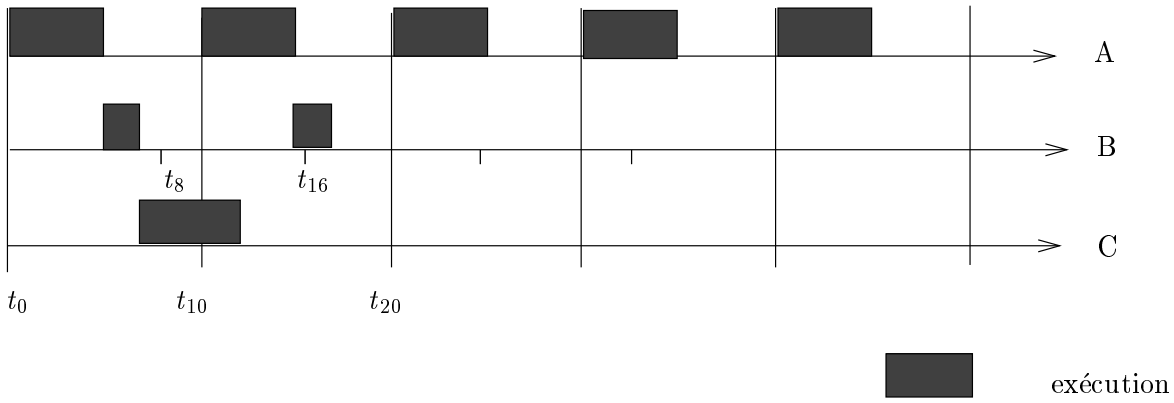


FIG. 10.5 – *Comportement temporel d'un schéma de synchronisation de tâches-modules*

ainsi assez simple de remonter au source, lorsque KRONOS fournit des résultats en termes d'états de l'automate temporisé.

10.3.2 Vérification d'un schéma de synchronisation de tâches-robot

L'exemple est fourni par l'environnement ORCCAD (Open Robot Controller Computer Aided Design), qui propose une approche intégrée, depuis une spécification de haut niveau jusqu'à l'implémentation. La notion de *tâche-robot* est centrale dans ORCCAD. Une tâche-robot est un ensemble de tâches temps-réel communicantes appelées *tâches-modules*. Pour spécifier complètement une tâche-robot, on associe à chacune de ses tâches-modules une durée de calcul (supposée connue) et une période d'activation, et on décide des protocoles de synchronisation utilisés entre les tâches (on considère en général des connexions binaires entre un producteur et un consommateur, et la lecture et/ou l'écriture peuvent être blocantes ou non).

La figure 10.4 donne une tâche-robot constituée de trois tâches-modules A, B et C. Le temps de calcul de A est de $5ms$, et cette tâche est activée toutes les $10ms$. Elle est synchronisée avec la tâche B par un protocole dit "asynchrone/synchrone" (dans la terminologie ORCCAD), c'est-à-dire que A évolue librement, alors que B est bloqué en attente d'une donnée produite par A. Le même type de synchronisation est utilisé entre B et C. La figure 10.5 donne le comportement temporel de la tâche-robot.

La figure 10.6 donne le programme ARGOS utilisé pour modéliser ce schéma de synchronisation.

La modélisation des tâches-robots en ARGOS temporisé, et l'utilisation de KRONOS, sont proposées comme une méthode de validation des schémas de synchronisation. En particulier, nous

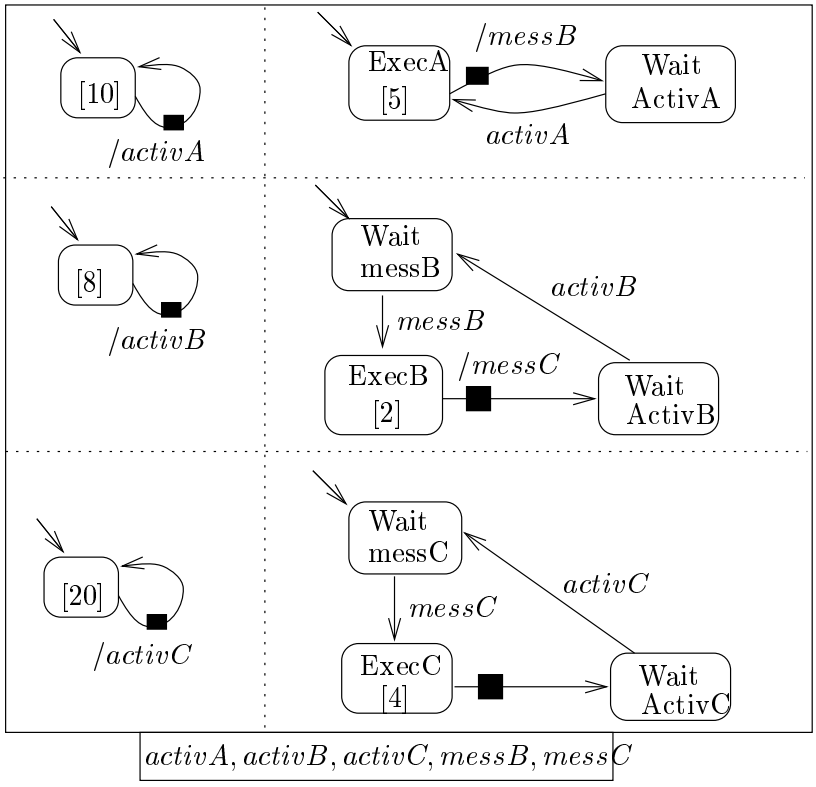


FIG. 10.6 – *Modèle* ARGOS *du schéma de synchronisation*

pouvons montrer automatiquement qu'un tel schéma est exempt de blocage temporel, autrement dit que les tâches-modules ont toujours fini leur exécution lorsque leur réactivation survient. Nous pouvons également exhiber les situations de blocage temporel, lorsqu'elles existent. Dans l'exemple proposé ci-dessus, un blocage apparaît, 16 ms après l'initialisation globale: la tâche B doit être réactivée, mais elle n'a pas même commencé son calcul, étant en attente d'activation par A.

10.4 Généralisation

Comme je l'exposais en introduction, nous avons, pour cette première extension d'ARGOS, redéfini la sémantique des constructions, qui expriment maintenant des compositions de "machines de Mealy temporisées". Cette nouvelle sémantique et sa propriété de compositionnalité s'inspirent bien sûr des travaux antérieurs, mais il apparaît finalement peu satisfaisant d'avoir à ré-écrire ainsi l'essentiel de la sémantique des opérations, tout en s'attachant à résoudre les nouveaux problèmes que pose l'extension temporelle.

Il est facile de se convaincre, en lisant la sémantique d'ARGOS temporisé, que les gardes temporelles et les affectations aux variables temporelles sont traitées indépendamment des gardes booléennes utilisées pour l'expression de la synchronisation usuelle.

Cette remarque suggère d'écrire la sémantique d'un ARGOS dans lequel on pourrait attacher

des informations quelconques aux états et transitions. Cette technique est utilisée pour décrire en ARGOS des systèmes dits *hybrides* et fait l'objet du chapitre 11.

Le cas temporisé n'est plus ainsi qu'un cas particulier du cas hybride. Nous abandonnons l'idée de prévoir dans le compilateur ARGOS un algorithme de compilation directe en automates temporisés. Ceux-ci peuvent être obtenus, depuis un programme qui utilise la macro-notation de temporisation ou des pragmas, grâce au prototype HARGOS décrit au chapitre suivant. La connexion à POLKA, qui peut réaliser des analyses approchées, est souvent nécessaire pour traiter des problèmes hors de portée de l'analyse exacte réalisée par KRONOS.

Chapitre 11

Argos hybride, un point d'entrée de l'outil Polka

Sources : *L'article [HMP95] expose un exemple d'utilisation de l'outil* HARGOS *obtenu en connectant une extension du compilateur* ARGOS *à l'outil* POLKA *de N. Halbwachs []. Nous reprenons cet exemple ici, en ajoutant une description du langage* ARGOS *hybride utilisé en entrée — non publiée pour l'instant — et quelques réflexions sur l'implémentation de la connexion.*

11.1 Motivations

Les motivations de ce travail sont doubles. Il s'agit, d'une part de généraliser le travail réalisé pour ARGOS temporisé ; d'autre part de permettre l'utilisation d'un langage structuré pour la description des systèmes hybrides sous forme de systèmes de transitions équipés de contraintes linéaires.

Le travail de réécriture de la sémantique d'ARGOS, pour tenir compte des contraintes temporelles associées aux états, fait apparaître une idée simple : ARGOS peut être considéré comme un jeu d'opérateurs pour combiner des automates, quelle que soit la nature des informations attachées aux états et transitions. En d'autres termes, il est possible de séparer complètement les aspects de calcul de la synchronisation par diffusion synchrone, des autres informations qu'on peut vouloir attacher aux états et transitions. Ces informations additionnelles sont introduites sous formes de *pragmas* (cf. chapitre 6), dont la sémantique est ignorée par le compilateur ARGOS.

En ce qui concerne les systèmes hybrides, la technique des pragmas d'ARGOS permet une description structurée plus facile à concevoir et manipuler qu'un automate plat. L'apport le plus intéressant concerne la vérification des programmes. En effet, la nature synchrone d'ARGOS permet, par l'utilisation d'*observateurs*, de ramener toute propriété de sûreté à l'accessibilité d'un état d'erreur défini à cet effet. Or l'outil POLKA est particulièrement approprié aux calculs d'accessibilité.

Notons également, en quelques mots, qu'un langage synchrone donne des descriptions réalistes du comportement de systèmes réactifs comme le contrôleur de passage à niveau décrit plus loin. En effet, on y distingue nettement les entrées et les sorties d'un système réactif, et les entrées doivent être acceptées telles que, le système ne maîtrisant que les sorties. Il existe en revanche de nombreuses descriptions du même genre de problèmes dans des langages ou formalismes à base de rendez-vous [], pour lesquelles les propriétés recherchées — par exemple : *la barrière est*

baissée lorsque le train passe — est assurée trivialement par un rendez-vous entre la barrière et le train : en quelque sorte, la barrière "arrête" le train si elle est ouverte !

11.2 Les systèmes hybrides et le modèle considéré

Dans les systèmes dits *hybrides*, des composants discrets contrôlent une activité continue. Dans le cas général l'activité continue devrait être décrite par des systèmes d'équations différentielles, qui constituent un modèle hors de portée des capacités d'analyse de propriétés par des méthodes informatiques. De nombreux travaux portent donc sur la définition de modèles moins expressifs que ce modèle général, mais pour lesquels on peut trouver des langages de propriétés analysables automatiquement.

Pour l'extension ARGOS hybride nous utilisons un modèle qui associe aux états et transitions d'un automate des systèmes de contraintes linéaires portant sur des variables numériques continues et sur leurs dérivées.

Plus précisément : les états portent des systèmes d'inéquations linéaires dans lesquels interviennent les dérivées des variables. Les dérivées sont supposées constantes dans chaque état, même si elles sont manipulées comme des constantes symboliques. Les conditions des transitions sont des systèmes d'inéquations linéaires sur les variables elles-mêmes, et les actions associées aux transitions sont des affectations des variables, pour lesquelles on ne s'autorise que des expressions linéaires. Ce modèle est analysable par des méthodes approchées développées par N. Halbwachs et implémentées dans l'outil POLKA. Ces méthodes consistent à calculer, pour chaque point de contrôle du programme, une approximation supérieure de l'ensemble des valuations de variables accessibles depuis une valuation initiale. Cette approximation est décrite par un polyèdre convexe.

11.3 Connexion ARGOS+POLKA : le prototype HARGOS

La connexion ARGOS/POLKA a été réalisée grâce à une légère modification du compilateur existant, qui permet de prendre en compte des *pragmas*. La nouvelle version de l'environnement ARGONAUTE, en cours de développement, intègre les extensions nécessaires à un traitement général des pragmas. La connexion à l'outil POLKA n'est plus qu'un cas particulier.

On utilise des pragmas d'états et de transitions, et un pragma global qui permet de définir les variables utilisées. Le compilateur ARGOS ignore ces pragmas et les transmet inchangés à POLKA, qui les analyse et peut alors y découvrir des erreurs de syntaxe. Pour éviter de reporter aussi tard la détection d'erreurs de syntaxe dans les pragmas, on peut imaginer que le compilateur ARGOS fasse appel à un programme séparé fourni par l'environnement aval. Ici, le concepteur de POLKA peut fournir un filtre qui renvoie, pour un pragma, un compte-rendu d'analyse. Le compilateur ARGOS, qui dispose de toutes les informations relatives au fichier source (numéro de ligne, ...) peut reporter ce compte-rendu assorti d'informations de position, et ne pas s'engager dans la phase de génération de code.

11.4 Exemple d'utilisation de HARGOS : le passage à niveau

On considère un passage à niveau sur une voie simple ou double, muni d'une barrière et activé lorsque le train écrase une pédale placée sur la voie. La région étudiée a pour longueur $D' = 1500m$, et la barrière est située à $D = 1000m$ de la pédale d'entrée (cf. figure 11.1). La vitesse maximale du train est de $S = 60m/s$. La barrière se ferme ou s'ouvre avec une vitesse

angulaire de $10^o/s$. Initialement, il n'y a pas de train dans la région étudiée, et la barrière est ouverte. A tout instant, ensuite, il y au plus un train sur chaque voie, dans chaque région.

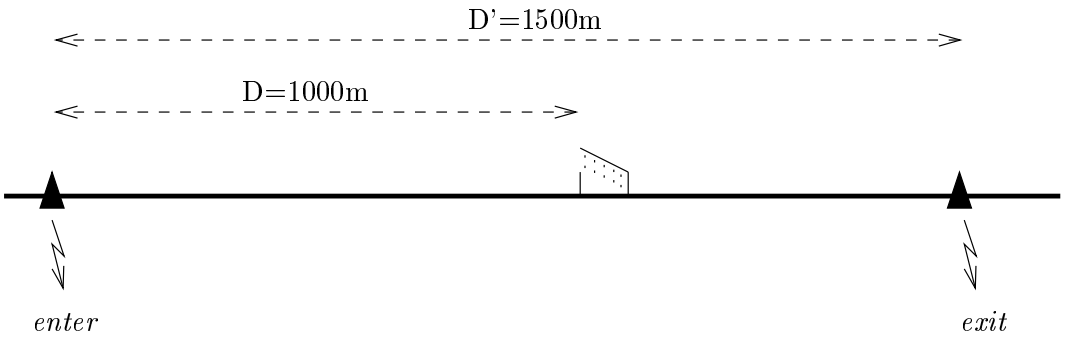


FIG. 11.1 – *Le problème du passage à niveau*

ARGOS hybride constitue une utilisation particulière de la technique des pragmas présentée ci-dessus. On associe aux états des invariants sous forme de systèmes d'inéquations linéaires portant sur les variables ou leurs dérivées, et aux transitions des conditions de franchissement données sous la même forme, et des affectations parallèles. Un pragma global permet de "déclarer" toutes les variables continues utilisées dans les invariants ou les gardes de transitions.

11.4.1 Première description en ARGOS hybride

Le système pourrait être décrit par un automate temporisé, par exemple en ARGOS temporisé (cf. chapitre 10). Il suffit pour cela de se ramener à une seule échelle de "temps" : le temps minimum qui séapre l'entrée du train dans la région et l'instant où il atteint la barrière est donné par $1000/60 = 16.67s$. Il lui faut en suite un temps supérieur à $(1500 - 1000)/60 = 8.34s$ pour quitter la région. La fermeture (ou l'ouverture) complète de la barrière prend $90/10 = 9s$.

Toutefois, nous donnons ici une description à temps multiforme, plus naturelle. La figure 11.2 donne le programme ARGOS hybride qui modélise le système complet : un train, la barrière (*gate*) et le contrôleur de passage à niveau.

En ARGOS hybride on peut associer aux états et transitions des automates de base des systèmes d'inéquations linéaires portant sur des variables continues ou leurs dérivées.

Le train est un système à trois états qui gère une variable continue x pour mesurer la distance du train au point d'entrée. On déclare, globalement, que $\dot{x} \leq 60$. Le train est initialement hors de la région ; la variable x est mise à zéro lorsque le train entre dans la région — signal *enter* émis vers le contrôleur ; tant que $x \leq 1000$, le train s'approche de la barrière, et l'atteint dès que $x = 1000$. Il est encore dans la région tant que $x \leq 1500$, puis la quitte dès que $x = 1500$, en émettant un signal *exit* vers le contrôleur.

La barrière gère une variable continue α pour mesurer son angle avec la position horizontale. La barrière est initialement ouverte ($\alpha = 90^o$). Dès qu'elle reçoit une commande *lower* elle commence à descendre avec une vitesse angulaire de 10 degrés par seconde ($\dot{\alpha} = -10$), jusqu'à être complètement fermée ($\alpha = 0$). Elle reste fermée ($\dot{\alpha} = 0$) jusqu'à ce qu'elle reçoive une commande *raise*. Elle s'ouvre alors avec une vitesse angulaire de 10 degrés par seconde ($\dot{\alpha} = +10$).

Le contrôleur reçoit les signaux *enter* et *exit* et compte les trains qui occupent la région. La variable k est une variable *discrète*, c'est-à-dire que sa valeur n'est modifiée que par les affectations explicites attachées aux transitions. Sa dérivée est globalement déclarée nulle. La région est vide de train lorsque $k = 0$. Les signaux *lower* et *raise* qui commandent la barrière lui sont envoyés quand la région devient occupée ou se libère, respectivement.

Si l'on considère un système simple à un train, il suffit de mettre en parallèle le contrôleur, la barrière, et un exemplaire du système train ci-dessus, en déclarant locaux les signaux *enter*, *exit*, *lower* et *raise*.

11.4.2 Compilation et analyse par POLKA

Le compilateur ARGOS produit, à partir de ce programme, un système de transitions à 15 états et 56 transitions. Les transitions sont étiquetées par des systèmes d'inéquations linéaires sont certains sont non satisfaisables.

La connexion à l'outil d'analyse de contraintes POLKA permet de supprimer ces transitions, et l'on obtient le système réduit suivant :

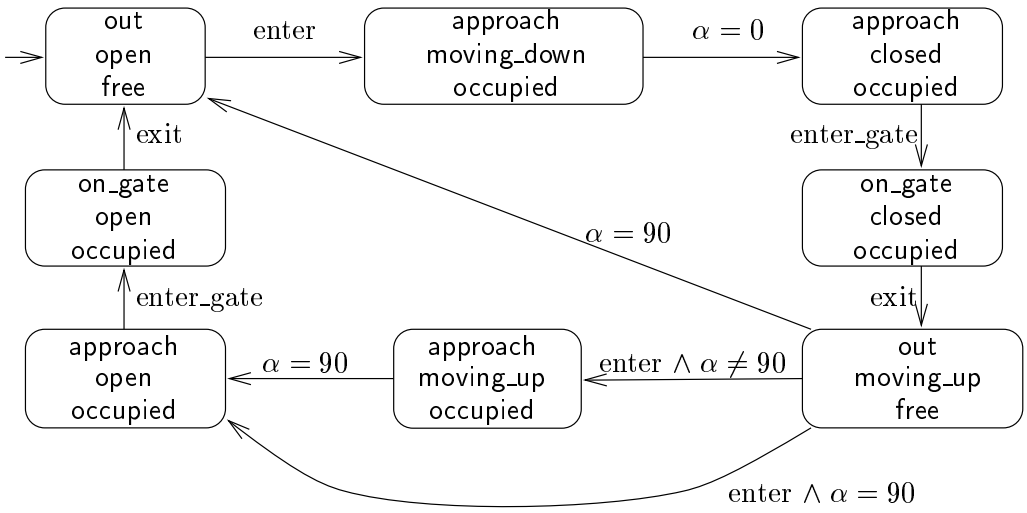


FIG. 11.3 – *Première description en* ARGOS *hybride, après réduction par* POLKA

Il semble donc que l'état *barrière levée/train présent* soit accessible. Cela révèle une erreur de conception : lorsque la barrière s'ouvre, elle ne tient pas compte des signaux *lower* qui peuvent lui parvenir, et continue donc à s'ouvrir même si un nouveau train survient. On corrige facilement la description de la barrière :

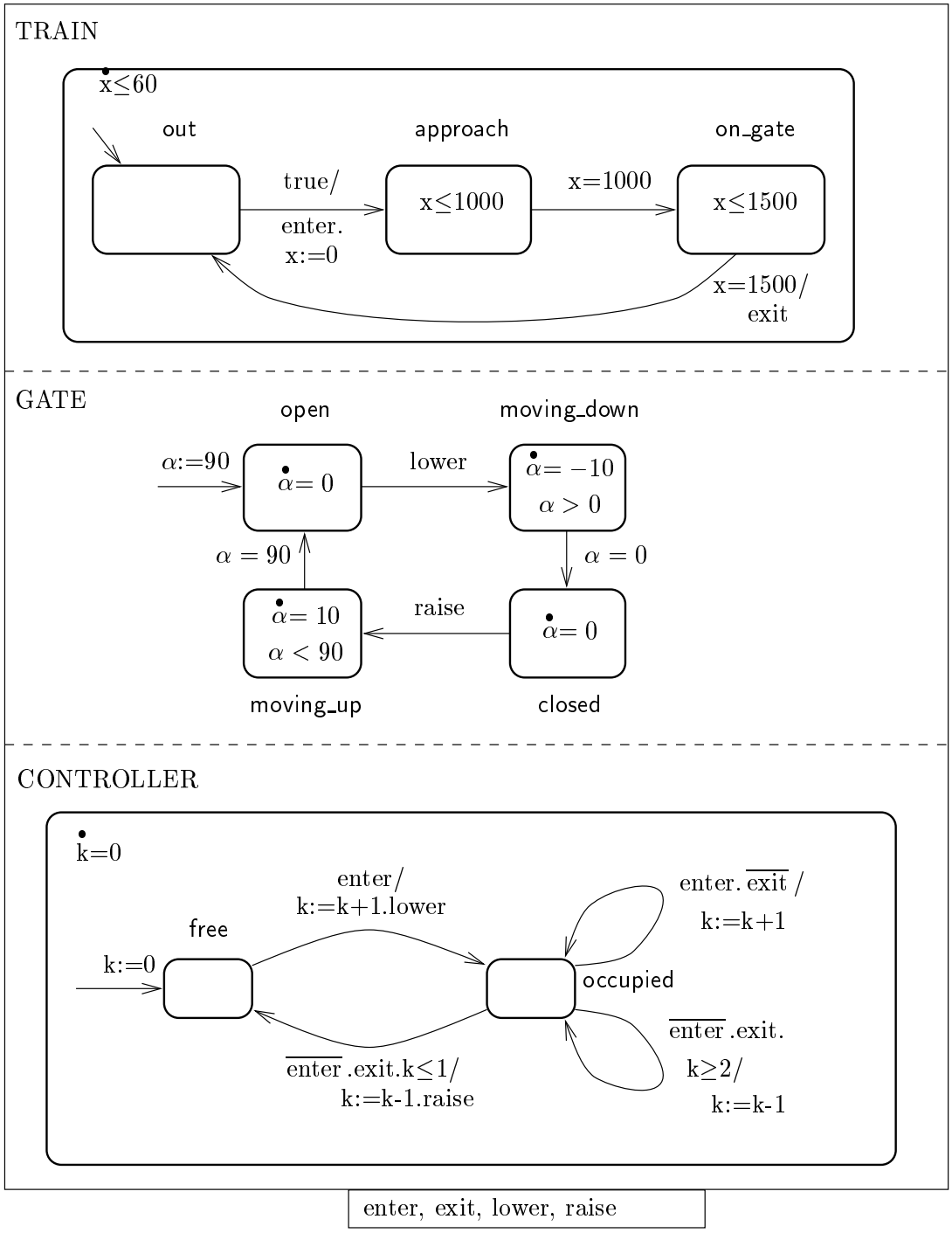


FIG. 11.2 – *Programme* ARGOS *hybride du passage à niveau*

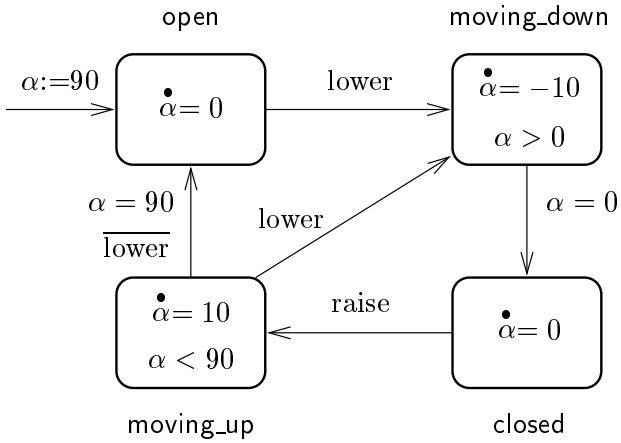


FIG. 11.4 – *Correction de la barrière*

Le système obtenu ne peut plus être décrit par un automate temporisé. En effet, lorsque le mouvement d'ouverture de la barrière est interrompu par une commande *lower*, la barrière doit redescendre de l'angle qu'elle avait atteint. Le temps nécessaire à ce mouvement de fermeture partiel n'est pas connu statiquement.

Si l'on compile le programme obtenu en remplaçant la description erronée de la barrière par la description ci-dessus, on obtient un programme ARGOS à 10 état et 22 transitions, dont POLKA ne conserve que la structure suivante, sur laquelle nous avons également reporté les résultats de l'analyse :

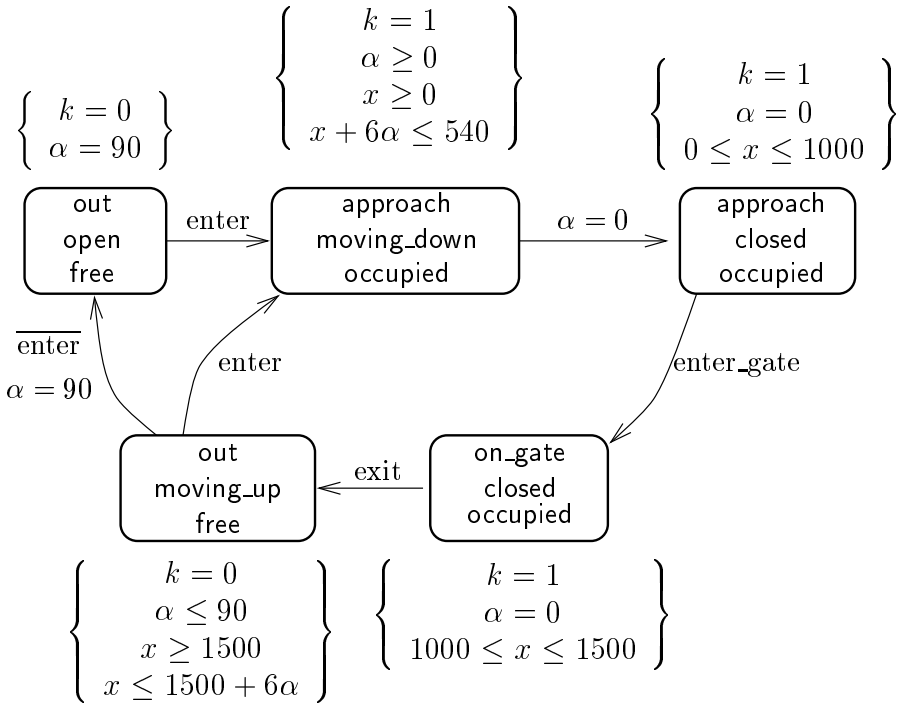


FIG. 11.5 – *Résultats de l'analyse pour un train*

11.5 Pour aller plus loin...

Analyse symbolique

L'outil POLKA peut être utilisé pour "découvrir" la distance minimum D entre la pédale d'entrée et la barrière, permettant d'assurer que la barrière est fermée quand le train passe. Il suffit de remplacer la distance 1000 par une constante symbolique D, et la distance 1500 par l'expression $D+500$. D est considéré par POLKA comme une variable de dérivée 0. L'outil montre que les états erronés (ceux pour lesquels le train passe uen barrière ouverte) ne sont accessibles que lorsque $D \leq 540$ (la barrière se ferme en 9 secondes, et, pendant cet intervalle de temps, le train parcourt au plus $60 \times 9 = 540m$.

Cette caractéristique de l'outil rend la connexion ARGOS+POLKA intéressante même lorsque les systèmes étudiés peuvent être décrits par des automates temporisés : l'outil KRONOS ne permet pas de manipuler les constantes de délai de manière symbolique.

Plusieurs trains

Pour prendre en compte plusieurs trains, il faut savoir distinguer les signaux *enter* et *exit* en provenance de trains distincts, afin de savoir tenir compte des occurrences simultanées de tels signaux. La figure 11.6 donne le nouveau contrôleur de barrière en ARGOS hybride. On y constate immédiatement la complexité des étiquettes. Une extension d'ARGOS pour prendre en compte des signaux valués (comme en ESTEREL) dont les valeurs peuvent être combinées en cas d'émission multiple réduirait quelque peu la description, mais les conditions à exprimer sont intrinsèquement complexes.

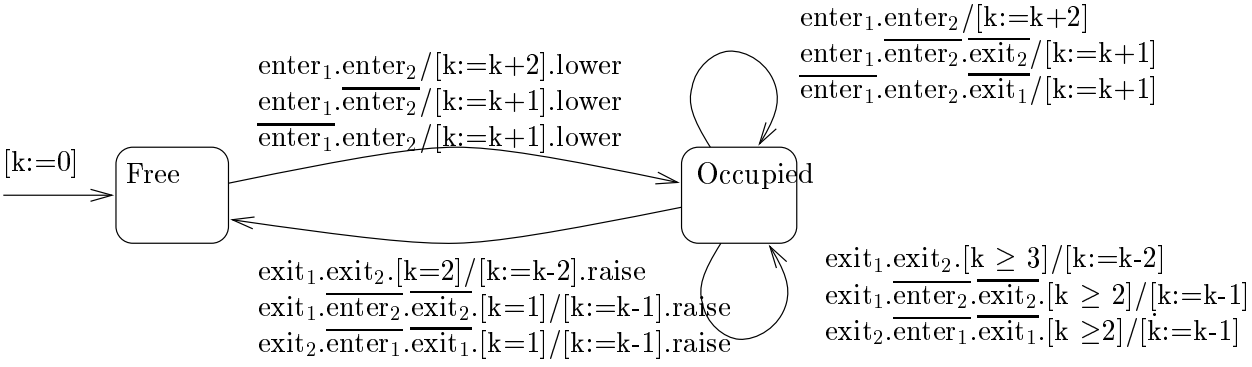


FIG. 11.6 – *Contrôleur de barrière pour deux trains*

Le compilateur ARGOS fabrique un automate à 37 états et 206 transitions, dont POLKA extrait une structure à 13 états. Les états problématiques ne sont pas accessibles.

Traitement du non déterminisme

Les descriptions de systèmes hybrides mais aussi, plus généralement, des systèmes équipés de pragmas, sont souvent non déterministes lorsqu'on s'abstrait des pragmas pour ne garder que la structure de contrôle ARGOS. Nous sommes là typiquement dans l'un des cas invoqués comme motivations du travail sur le non déterminisme dans les langages synchrones, exposé au chapitre 4. Je n'y reviendrai pas ici.

Quatrième partie

Environnement de programmation

Chapitre 12

Edition graphique et débogage symbolique d'argos

Sources : *Les réflexions présentées ici doivent beaucoup aux discussions avec Philippe Schaar, lors de son année de stage à Vérimag, et avec Vincent Quint, du projet INRIA Opéra, au sein duquel a été conçu et développé l'outil Grif. L'expérience menée au cours de l'année 1993 autour de l'implantation d'un éditeur* ARGOS *en Grif est mentionnée dans [QV94]. Le travail de P. Schaar est décrit dans [Sch94].*

12.1 Présentation du problème

Le problème de l'édition graphique d'ARGOS s'est posé très tôt. A la fin de ma thèse, il existait un environnement de programmation complet pour ARGOS, comprenant un éditeur graphique syntaxique et la première version du compilateur, et connecté à l'outil d'analyse ALDÉBARAN []. L'éditeur avait été entièrement spécifié en LPG (un langage de types abstraits algébriques []) et programmé en XWindows-10, par un groupe de 4 étudiants du DESS de Génie Informatique de l'Université Joseph Fourier. Parallèlement à ce développement expérimental, des réflexions étaient menées sur les fonctionnalités nécessaires dans un éditeur syntaxique de langage graphique, renforcées plus tard par les commentaires de Muriel Jourdan confrontée à l'utilisation de l'éditeur pour décrire le premier "gros" programme ARGOS.

L'éditeur était conçu comme un simple programme d'application au-dessus d'une bibliothèque de manipulation d'arbres abstraits et de représentation graphique des objets du langage. Une utilisation non interactive de la même bibliothèque avait permis d'obtenir un débogueur symbolique[1] d'ARGOS, en modifiant le compilateur de manière à ce qu'il associe, aux états et transitions de l'automate global d'un programme, les ensembles d'états et de transitions des automates de base dont ils proviennent. On pouvait donc exécuter l'automate global en observant, sur la forme graphique du programme, les états actifs et les transitions déclenchées.

Le résumé des reflexions et commentaires tirés de ces expériences tient en quelques points importants. Tout d'abord, pour réaliser un éditeur de langage graphique, il a deux solution extrêmes, entre lesquelles on peut imaginer toute une variété de solutions mixtes :

Analyse syntaxique 2D : on peut utiliser un éditeur graphique général, et réaliser une analyse

1. Au sens où `gdb` est un débogueur symbolique de C, par exemple ; "symbolique" faisant ici référence à la table des *symboles* du programme, conservée au cours de la compilation comme information de retour au source.

de la figure pour y repérer les objets du langage (c'est exactement l'approche suivie pour les langages séquentiels, où l'on invente une syntaxe concrète à base de séparateurs et de mots-clés pour représenter linéairement, sans perte d'information, une structure typiquement arborescente. L'analyse syntaxique consiste alors à reconstruire l'arbre abstrait à partir du texte linéaire). Il est préférable d'utiliser un éditeur qui fournit des objets de base comme les rectangles et les lignes, plutôt qu'un éditeur de bitmap. Si tous les éléments de syntaxe concrète du langage correspondent à des objets graphiques connus de l'éditeur, une partie du travail de reconnaissance est épargné. Reste toutefois à repérer l'organisation de ces objets (typiquement, dans un éditeur d'automate, quel critère de proximité peut-on utiliser pour reconnaître l'association flèche-texte qui définit une transition ?). Cette approche, qui peut sembler sans espoir, a pourtant été utilisée dans d'autres contextes, en particulier avec l'éditeur Idraw de l'INRIA qui produit du postscript commenté par les types d'objets. D'autre part il existe des travaux théoriques sur la généralisation des algorithmes d'analyse syntaxique de texte à des sources 2D. Voir par exemple [CTC91, CTC90], dans lesquels on utilise une relation spatiale entre objets.

Edition syntaxique : on peut développer un éditeur syntaxique, basé sur un outil de dessin. On se trouve alors confronté à trois types de problèmes :

- des problèmes de dessin plus ou moins interactif : typiquement, dans un éditeur d'automates, il est bien agréable de disposer d'un placeur automatique de transitions, et on aimerait que le déplacement de la boîte qui représente un état s'accompagne d'un déplacement harmonieux des flèches représentant les transitions qui y sont attachées

- des problèmes classiques de manipulation d'arbre abstrait : doit-on imposer une construction ascendante ou descendante, quelles opérations de suppression de sous-arbre peut-on autoriser...? sont quelques unes des questions qui se posent au concepteur.

- des problèmes de choix du degré d'analyse garanti par l'éditeur : peut-on se contenter d'un éditeur qui ne gère que les aspects de syntaxe hors-contexte, ou doit-on également garantir une analyse des aspects de syntaxe contextuelle (ou sémantique statique) ? Dans ce dernier cas, à quels blocs s'applique l'analyse, et quand ? (La réponse "*partout, tout le temps*" conduit à définir un éditeur dans lequel tout objet à peine ébauché doit être correctement constitué avant de pouvoir être utilisé comme composant d'un objet plus complexe. C'est un comportement extrêmement contraignant, pour ne pas dire pénible).

Nous n'avons jusque là considéré que les aspects de création et modification de programmes. Un autre aspect important concerne les choix de présentation des programmes. En effet, comme exposé dans [Mar91] pour ARGOS, un langage graphique est intrinsèquement peu concis[2]. Il faut donc prévoir des techniques de présentation partielle des programmes — par exemple en ne montrant pas systématiquement le contenu des états raffinés — et imaginer des opérations de manipulation de la présentation — zoom sur un composant raffinant, etc.

2. Contrairement à l'idée reçue qui veut qu'*un dessin vaille mieux qu'un long discours*

12.2 Première tentative : Andromède

La première réalisation d'un éditeur ARGOS est un éditeur syntaxique. Nous avons en quelque sorte reinventé les problèmes d'édition d'arbre abstrait — et leurs solutions les plus immédiates —, au cours du développement, par manque de culture dans ce domaine. Le prototype Andromède propose une édition ascendante — avec placeur automatique des transitions des automates et des composantes d'une mise en parallèle — couplée à des vérifications de sémantique statique incrémentales. Pour modifier l'étiquette d'une transition appartenant à un automate pris dans une construction complexe à plusieurs étages d'opérateurs, il faut "déconstruire" l'arbre afin d'isoler l'automate, le modifier, puis reconstruire l'arbre. Toute modification locale, comme l'insertion d'un opérateur unaire portant sur un sous-arbre, est réalisée par déconstruction et reconstruction. Cet aspect, à lui seul, rend le prototype inutilisable en pratique.

Et pourtant... ce problème d'édition graphique est mis de côté temporairement, et on définit une syntaxe textuelle très simple pour ARGOS, afin de pouvoir utiliser le compilateur et continuer à travailler sur la sémantique en disposant d'un outil.

12.3 Le prototype Aglae : éditeur ARGOS en Grif

La deuxième tentative conduit au prototype Aglaé, conçu et développé par Philippe Schaar dans le cadre de son mémoire de thèse CNAM en 1993 [Sch94]. Confronté au problème dans son entier, et à une démonstration de l'éditeur Andromède, encore exécutable à l'époque, P. Schaar a fait preuve d'une remarquable et salutaire curiosité. Après avoir trouvé, compris et testé un certain nombre de bibliothèques, "méthodes" ou environnements de conception d'interfaces homme/machine — le chapitre 4 de son mémoire de thèse est un bon résumé de ce tour d'horizon — il a finalement suggéré d'utiliser Grif [], développé au sein du projet INRIA Opéra (Outils Pour les documents Electroniques, Recherche et Applications). Plus exactement, il a suggéré une interview des chercheurs du projet Opéra, sur les aspects d'édition "intelligente" d'arbres abstraits. Cette rencontre s'est concrétisée par la décision d'implanter l'éditeur ARGOS en Grif, en utilisant l'interface programme-application (API) en cours de définition pour Grif.

12.3.1 Bref aperçu de Grif

Grif est un outil d'édition de documents structurés, qui propose de définir séparément la structure *logique* du document et ses différentes *présentations*. La structure logique est décrite par une grammaire hors-contexte équipée de liens hyper-texte. Une présentation est décrite en associant à chaque élément de la structure logique un ensemble de règles définissant son aspect graphique. Plusieurs présentations différentes peuvent être définies, pour la même structure logique. L'édition est contrôlée par la structure: toute action de l'utilisateur, sensible sur une des présentations du document, est traduite en termes de la structure logique, et peut-être répercutée sur les autres présentations. L'édition est générique, au sens où l'éditeur propose des actions de base de manipulation des arbres sous-jacents, comme la création, le déplacement ou la suppression de sous-arbres. L'éditeur propose aussi une opération de "copier/coller" intelligente, c'est-à-dire respectant les règles de définition de la structure logique.

12.3.2 Aglaé

Aglaé s'appuie sur Grif pour définir un éditeur ARGOS mi textuel, mi graphique. Les principes de base sont les suivants : la description des automates n'est pratique que si l'on dispose de l'édition graphique. Toutefois l'édition des informations textuelles associées (étiquettes de transitions, nom et commentaires associées aux états) est plus agréable dans un éditeur de texte. La structure générale des processus s'édite plus facilement sous une forme textuelle, qui montre les imbrications par des indentations de manière plus concise que la syntaxe graphique à deux dimensions. La présentation graphique des processus composés est entièrement calculée.

L'éditeur Aglaé fournit des outils de classement des automates et des processus définis dans le programme, de recherche par nom, de renommage global, etc.

Bien que l'implantation effective se soit heurtée aux performances encore insuffisantes de l'API de Grif en ce qui concerne les objets graphiques, le travail de définition d'Aglaé a permis d'établir le cachier des charges de ce que serait un véritable éditeur mixte graphique/textuel pour un langage comme ARGOS. Sans doute faudrait-il reprendre l'étude aujourd'hui, avec la version actualisée de Grif.

12.4 Travaux de même nature

12.4.1 Autograph

Valérie Roy (CMA - Ecole de Mines, Sophia Anitpolis) travaille depuis presque 10 ans sur l'outil AUTOGRAPH [RS89, RdS90, Roy90] de manipulation des représentations graphiques de systèmes de transitions ou de réseaux de processus. La version actuelle offre un algorithme de placement automatique. L'idée la plus intéressante d'AUTOGRAPH réside dans le mécanisme de placement guidé d'un système de transitions : l'outil propose les états par "tranches" successives depuis l'état initial. Ainsi on place d'abord les états immédiatement successeurs de l'état initial, puis on explore chacun d'entre eux et on place ses propres successeurs, etc...

Nous avons envisagé dès le début l'utilisation d'AUTOGRAPH pour l'édition de programmes ARGOS. Il nous a paru difficile d'adapter l'outil à la manipulation des structures de programmes.

12.4.2 Arged (GMD, Sankt Augustin, projet Synchronie)

Leszek Holenderski, qui travaille dans l'équipe d'Axel Poigné (GMD, Sankt Augustin) sur l'intégration des langages ESTEREL, LUSTRE et ARGOS dans un environnement unique, s'est lancé dans le développement d'un éditeur ARGOS en TCL/TK, nommé Arged.

Outre les problèmes de performance liés à une première version interprétée — qui seraient sans doute "aisément" résolus par une programmation mixte TK/C, en version compilée — le prototype présente les défauts d'Andromède : la manipulation des arbres abstraits est extrêmement contraignante, bien que Leszek ait défini tout un ensemble de manipulations "naturelles" comme l'insertion d'un opérateur d'encapsulation en position de noeud interne.

Dans la version la plus récente, on retrouve une partie des manipulations d'arbres abstraits qui avaient été proposées par Philippe Schaar pour Aglaé. On peut ainsi sélectionner un sous-arbre de l'arbre abstrait d'un processus, et lui appliquer des opérations comme l'ajout d'un opérateur unaire, ou l'ajout d'un composant parallèle. Toutefois il manque encore une opération de copier/coller respectant la structure typée de l'arbre abstrait, pour pouvoir réaliser des constructions descendantes ou ascendantes.

L. Holenderski a également prévu le retour au source, et le même outil peut être utilisé pour simuler le comportement d'un programme en montrant les états courants et les transitions tirées.

12.4.3 Editeur des SyncCharts (Université de Nice)

Charles André (Université de Nice) travaille sur une extension d'ESTEREL par des constructions hiérachiques héritées des Statecharts (dans le style d'ARGOS) et par des constructions héritées du Grafcet. L'ensemble est traduit en ESTEREL. Il existe un éditeur pour ce nouveau langage à syntaxe graphique, également programmé en TCL/TK, mais beaucoup plus efficace qu'Arged cité ci-dessus.

Chapitre 13

Compilation et connexion aux outils

Sources : *Documentation du nouveau compilateur. Les versions antérieures ont été décrites dans[Jou91, Jou94, MV92b]*

13.1 Schéma général du compilateur, formats et étapes

La figure 13.1 donne le schéma de la structure interne du nouveau compilateur. On y trouve les différentes formes intermédiaires et les différents chemins qui conduisent du format source ARGOS à un programme exécutable ou à un modèle analysable. Certaines phases ne sont pas encore implantées ou pas encore intégrées dans ce nouveau cadre, mais ont été testées indépendamment dans des prototypes (c'est le cas par exemple de la traduction ARGOS vers DC).

Analyse

Partant d'un programme ARGOS source (un texte à la syntaxe décrite au chapitre 6), la phase d'analyse syntaxique produit un arbre abstrait, puis un abre abstrait correct vis-à-vis des types et de l'utilisation des noms (déclarations, ...).

Traitement du non-déterminisme

L'étape suivante consiste à obtenir un arbre abstrait décrivant un ensemble de processus déterministes. Deux cas sont traités :

- Si l'on n'accepte pas le non déterminisme intrinsèque en entrée, certains cas de non déterminisme peuvent être traités comme des abréviations (voir chapitre 6). Les autres provoquent des erreurs de compilation.

- Si l'on accepte le non déterminisme intrinsèque en entrée, on applique la phase d'introduction des oracles décrite au chapitre 4 pour obtenir un programme déterministe et pouvoir enchaîner les phases suivantes.

Générations d'équations puis de programmes DC

A partir d'un arbre abstrait correct décrivant un ensemble de processus déterministes, on génère des équations booléennes (structure du format DC). On peut suivre différents chemins

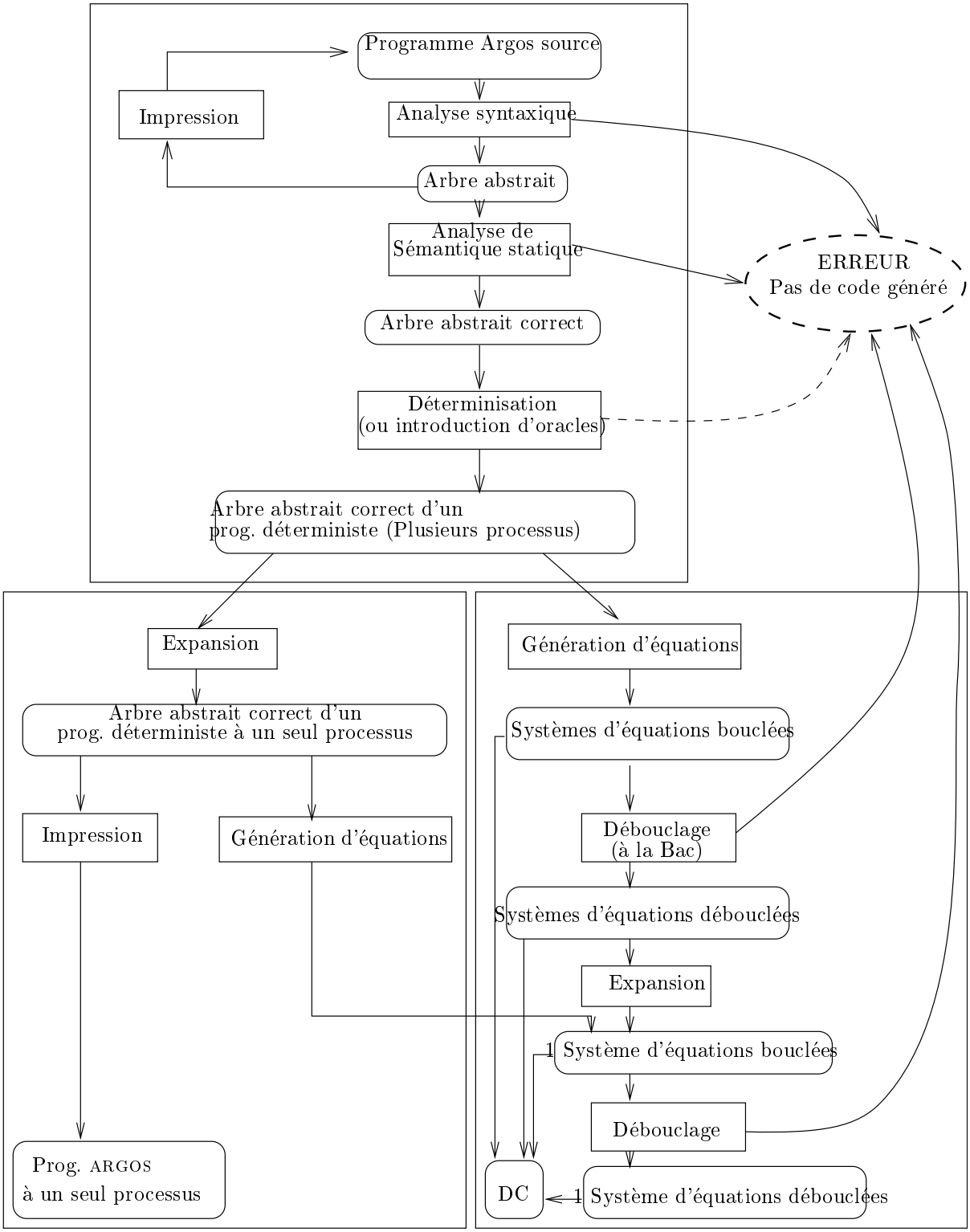


FIG. 13.1 – *Phases et formats intermédiaires de la chaîne de compilation*

selon que l'on expanse le programme ARGOS préalablement ou non.

- En cas d'expansion préalable, la génération produit un système d'équations éventuellement bouclées (c'est-à-dire un noeud DC, aux équations bouclées)

- Si l'on n'expanse pas le programme ARGOS, la génération d'équations produit un noeud DC par processus ARGOS, aux équations potentiellement bouclées également.

Chaque système d'équations peut être débouclé indépendamment du contexte dans lequel le processus correspondant est utilisé. Ce débouclage peut produire une erreur de compilation s'il est impossible d'obtenir un ensemble d'équations débouclées. L'expansion au niveau des noeuds DC peut réintroduire des boucles, que l'on doit de nouveau éliminer.

Si l'on expanse le programme ARGOS préalablement, les boucles éventuelles d'un processus sont reproduites en autant d'exemplaires que d'appels de ce processus, et le travail de débouclage est donc dupliqué (d'autre part il est plus facile, et plus clair, de reporter l'impossibilité de déboucler sur un programme non expansé). Un débouclage local à chaque processus, avant expansion, minimise ce travail. On n'échappe pas, pourtant, à une deuxième phase de débouclage, une fois les équations expansées, mais elle ne porte plus que sur un petit ensemble de signaux locaux (et on sait qu'une erreur découverte à cette étape provient de la structures des appels de processus, et non de la structure d'un processus particulier).

La génération de programmes au format DC est une simple étape d'impression des systèmes d'équations manipulés jusque là. Puisqu'on accepte les boucles de définition en DC, il est possible de produire des programmes DC à n'importe quelle étape.

Obtention d'automates explicites

L'obtention d'un automate explicite (format OC d'automates interprétés pour la connexion au générateur de code OCC, format TARGOS d'automates booléens à pragmas pour la connexion à POLKA, format AUT d'automates pour la connexion à ALDÉBARAN, etc.) peut être réalisée à partir d'un système d'équations non bouclées.

On peut aussi espérer des outils de génération d'automates explicites à partir de DC.

Traitement des pragmas

Le mécanisme de composition des pragmas a été décrit au chapitre 6. Il permet de conserver, dans la structure des pragmas, le maximum d'information sur la structure des processus.

Deux aspects doivent être étudiés pour l'implantation de ce mécanisme :

- Le comportement des pragmas lors de l'expansion des processus : on décide que les pragmas de processus sont hérités par les objets du processus, de la même manière que le pragma d'un état est hérité par les objets du processus qui raffine cet état (voir l'exemple au chapitre 6).

- Un mécanisme permettant de transmettre les pragmas ARGOS aux variables des programmes DC qu'on obtient en bout de chaîne : la question se pose puisque, lors de l'étape de codage des programmes ARGOS en équations booléennes — que ce soit après ou avant l'expansion — on perd les structures syntaxiques auxquelles les pragmas étaient attachés. Si chaque état a été codé par une variable, le pragma d'état peut devenir le pragma de la variable. Pour les transitions, rien de tel n'est possible. Une solution est d'associer à chaque

transition syntaxique du programme ARGOS un signal de sortie. Ainsi la variable associée à ce signal est vraie exactement lorsque la transition est tirée. Le pragma de transition peut être attaché à cette variable.

Autres connexions

Les connexions diverses qui avaient été implantées dans les premières versions du compilateur sont toujours possibles à partir d'un programme ARGOS expansé, c'est-à-dire à un seul processus. C'est le cas de la connexion à MEC [ABC94] et TOUPIE [Rau92]. Il faudrait réétudier la question pour une génération avant expansion.

La connexion à KRONOS est obtenue comme un cas particulier de la connexion à POLKA.

Cinquième partie

Recherche et enseignement

Chapitre 14

Automates et programmation par événements en DEUG

Sources : *Ce chapitre est issu du travail réalisé avec P.-C. Scholl, M.-C. Fauvet et F. Lagnier pour l'enseignement de l'informatique en* DEUG *scientifique à Grenoble. La première partie, qui traite de l'étude des algorithmes de parcours séquentiels, est publiée dans [SFLM93], et effectivement enseignée. Suivent deux propositions, qui découlent assez naturellement de ce travail, et qu'il serait intéressant d'expérimenter : une approche de la preuve d'algorithmes séquentiels à travers leurs représentations en automates, et la description d'algorithmes de parcours par des programmes* ARGOS.

14.1 Motivations

Le but de ce chapitre est de montrer le lien étroit entre le domaine des systèmes réactifs et une application des techniques et outils définis dans ce cadre à des activités d'enseignement de premier cycle. La place manque pour replacer le chapitre dans le contexte global de l'enseignement de la filière considérée, et il sera donc peut-être difficile de juger du contenu sur des critères pédagogiques. L'essentiel du raisonnement, et la méthode consistant à définir des congruences par le biais d'abstractions, sont déjà développés dans le chapitre 3, à propos d'un modèle général permettant d'exprimer plusieurs modes de communication synchrone, afin de les comparer.

Les paragraphes 14.2 et 14.3 proposent une approche pour l'étude des algorithmes itératifs dans laquelle l'accent est mis sur la découverte des variables nécessaires à l'itération. Dans le cas où ces variables ont un domaine fini — de taille raisonnable — nous proposons au paragraphe 14.4 de représenter les algorithmes de parcours de séquence par des automates. Cette idée a deux aspects : conception des algorithmes sous forme d'automate, pour s'abstraire de la mécanique d'accès à la séquence et se concentrer sur la mise à jour de la mémoire ; preuve d'algorithmes représentés sous forme d'automates. Enfin nous proposons au paragraphe 14.5 d'utiliser les constructions d'ARGOS pour décrire des algorithmes itératifs.

14.2 Etude des algorithmes itératifs

La plupart des ouvrages d'initiation à l'algorithmique traitent de la construction des itérations en insistant sur la correction et sur la terminaison. C'est l'occasion d'introduire les invariants, les axiomes de la logique de Hoare, et de raisonner formellement sur des programmes. D'un point

Elément : un type { *Le Type des éléments de la séquence* }
Démarrer : une action
{ *précondition : vrai*
postcondition : 1) la séquence est vide, FinDeSéq *est vrai,* EC *n'a pas de sens ; OU 2) la séquence n'est pas vide,* EC *donne accès au premier élément* }
Avancer : une action
{ *précondition :* FinDeSéq *est faux.*
postcondition : 1) on était sur le dernier élément de la séquence. FinDeSéq *est devenu vrai,* EC *n'a plus de sens ; OU 2) on n'était pas sur le dernier élément :* EC *donne accès à l'élément suivant.* }
FinDeSéq : $\longrightarrow$ un booléen
{ *vrai lorsque la fin de la séquence est atteinte.* }
EC : $\longrightarrow$ un Elément
{ EC *donne l'élément courant de la séquence.* }

FIG. 14.1 – *Une machine d'accès séquentiel*

de vue méthodologique, on suggère également que la construction d'une itération commence par l'écriture de son invariant.

La question du choix des variables de l'itération est, en revanche, rarement abordée. Cela constitue pourtant un problème parfois difficile. D'autre part on juge souvent les algorithmes écrits par les étudiants sur des critères plus ou moins conscients d'élégance et de simplicité, et un mauvais choix des variables d'une itération conduit souvent à écrire un algorithme peu élégant, au sens où certaines informations manipulées sont inutiles au traitement demandé.

Ramenons tout de suite ces considérations à l'exemple typique "*compter les 'LE' dans un texte*". Les algorithmes sont donnés dans la notation algorithmique adoptée dans [SFLM93]. Il s'agit d'un pseudo-Pascal dont les mots-clé sont en français. On utilise un modèle d'accès séquentiel du type avec marque de fin (les chaînes de caractères en C par exemple), et on fournit les primitives suivantes données figure 14.1.

Exemple 14.1 : Un algorithme itératif : Compter les 'LE'
Version 1 :

```
Elément : le type caractère
nble : un entier ≥ 0 ; CarPrec : un caractère          { Le caractère précédent }
Démarrer ; nble ⟵ 0
CarPrec ⟵ 'X'                                          { Un caractère différent de 'L' }
tant que non FinDeSéq
   si CarPrec = 'L' et EC = 'E' alors nble ⟵ nble + 1
   CarPrec ⟵ EC ; Avancer
Ecrire (nble)
```

Version 2 :

```
PrecEstUnL : un booléen                                { La "mémoire" booléenne }
Démarrer ; nble ⟵ 0 ; PrecEstUnL ⟵ faux
tant que non FinDeSéq
   si PrecEstUnL et EC = 'E' alors nble ⟵ nble + 1
   PrecEstUnL ⟵ EC = 'L' ; Avancer
Ecrire (nble)
```

□

Dans la version 1, on mémorise le caractère précédent, et c'est une information trop riche pour le traitement demandé. On s'en rend compte en observant comment est utilisée la variable CarPrec : elle n'apparaît que dans un test CarPrec = 'L'. Il suffit donc de mémoriser le booléen CarPrec = 'L'. Ce choix donne la deuxième version de l'algorithme, que beaucoup jugeront plus élégante. Un critère objectif d'élégance, dans ce cas précis, est le caractère minimal de la mémoire utilisée. La mémoire n'est pas minimale s'il en existe une abstraction qui est encore suffisante pour le traitement demandé ("abstraction" est pris au sens : image par une fonction non injective).

Il est intéressant de placer le discours à ce niveau, parce qu'aucun critère quantitatif de place mémoire ne peut justifier qu'on préfère un booléen à un caractère : les booléens, aussi bien que les caractères, sont en général codés sur un octet. On peut éventuellement discuter des temps d'exécution respectifs des instructions d'assembleur qui ont toutes chances d'être produites par la compilation de "si CarPrec = 'L'" et "si PrecEstUnL" [1], mais le véritable argument est plus fondamental.

14.3 Un schéma général d'itération

L'étude détaillée des exemples où ce type de raisonnement sur la mémoire nécessaire intervient, nous a conduits à proposer un schéma général d'itération dans lequel on distingue des *variables d'historique* et des variables *résultat*. L'application systématique de ce schéma doit permettre de raisonner plus facilement sur la mémoire nécessaire au traitement.

14.3.1 Le schéma (figure 14.2 ci-dessous)

Le type typR, la fonction AccumulationDe, la constante ResInit et la variable R sont introduits pour décrire le calcul du résultat. Le type Histoire, la constante HistoireInitiale, la variable H et la fonction MiseAJour permettent de manipuler l'*histoire*, c'est-à-dire l'information qu'on a choisi de mémoriser pour permettre le calcul du résultat. Dans les commentaires, on fait référence à la partie gauche de la séquence, notée *pg*, déjà parcourue et traitée, qui représente la mémoire maximale que l'on pourrait conserver durant le parcours. Toutefois cette partie gauche n'est pas mémorisée. L'*histoire* H est une abstraction de la mémoire maximale, définie par une fonction non injective notée *Ab*, qui doit avoir la propriété suivante, dite *propriété de mise à jour* : $Ab(\mathsf{pg} \bullet \mathsf{EC}) = \mathsf{f}\ (Ab(\mathsf{pg}),\ \mathsf{EC})$. En d'autres termes, la nouvelle histoire $Ab(\mathsf{pg} \bullet \mathsf{EC})$ doit pouvoir s'exprimer en termes de $Ab(\mathsf{pg})$ (l'histoire conservée jusqu'ici) et EC (l'élément courant) seulement. On exprime ainsi que l'abstraction choisie représente une information suffisante au traitement à réaliser, et on peut donc écrire H $\longleftarrow$ MiseAJour (H, EC).

1. en 68000, il faut comparer, à mode d'adressage égal, les temps d'exécution des instructions `cmp` et `tst`, qui ne diffèrent pas sur des octets ; en SPARC la distinction n'existe pas, et ces deux opérations sont réalisées par des soustractions.

```
Elément, Histoire, typR : des types
AccumulationDe : une Histoire, un Elément, un typR  ⟶ un typR
ResInit : un typR { le résultat dans le cas où la séquence donnée est vide }
HistoireInitiale : une Histoire
MiseAJour : une Histoire, un Elément  ⟶ une Histoire
H : une Histoire                                   { Abstraction de la partie gauche }
R : un typR                                        { Le résultat en cours de calcul }

Démarrer ; H ⟵ HistoireInitiale ; R ⟵ ResInit
{ H = Ab ([ ]). ([ ] représente la séquence vide). }
tant que non FinDeSéq
    { Invariant : H = Ab (pg). (pg est la partie de la séquence déjà parcourue). }
    R ⟵ AccumulationDe (H, EC, R)
    H ⟵ MiseAJour (H, EC)
    { H = Ab (pg • EC). (• représente l'ajout d'un élément à droite) }
    Avancer
```

FIG. 14.2 – *Schéma de parcours séquentiel avec variables d'*historique

14.3.2 Utilisation du schéma pour raisonner sur la mémoire nécessaire

Lorsque le traitement à réaliser est complexe, et qu'il est difficile de choisir la mémoire, on peut toujours commencer par écrire un algorithme où la mémoire est maximale. En analysant ensuite, dans l'expression de la fonction AccumulationDe, comment est utilisée l'histoire, on peut déterminer une abstraction suffisante.

Exemple 14.2 : Un algorithme de parcours : majuscules et minuscules
On considère une séquence de lettres. On désire construire une séquence modifiée, dans laquelle les lettres de rang pair sont reproduites en majuscules, et les lettres de rang impair en minuscules. On suppose l'existence de deux fonctions de traduction Maju et Minu. □

Si l'histoire utilisée est la partie gauche de la séquence (la mémoire maximale), on a : ResInit = [ ] (la séquence vide) et
AccumulationDe (H, EC, R) = R • (si pair(longueur (H)) alors Minu(EC) sinon Maju(EC))

Autrement dit, pour ajouter une lettre au résultat R en cours de construction, il suffit de savoir déterminer pair(longueur (H)). On choisit donc comme fonction d'abstraction : *Ab* (s) = pair (longueur (s)) où s est une séquence. Cette fonction d'abstraction a la propriété requise pour la mise à jour, puisque : *Ab*(s • e) = pair (longueur (s • e)) = pair (longueur (s) + 1) = non pair (longueur (s)) = non *Ab*(s).

Avec cette abstraction booléenne, on peut instancier le schéma général :

```
Histoire : le type booléen
ResInit : la séquence [ ]
HistoireInitiale : le booléen vrai
MiseAJour (H, EC) = non H
AccumulationDe (H, EC, R) = R • (si H alors Minu(EC) sinon Maju(EC))
```

14.3.3 Abstractions trop fortes

Pour prouver P par récurrence, il peut être nécessaire de prouver une propriété plus forte que P, parce que P elle-même n'est pas inductive.

De même, dans la démarche décrite ci-dessus, l'examen de l'utilisation qui est faite de H dans la fonction d'accumulation (le calcul du résultat) peut conduire à une abstraction trop forte. Dans ce cas la fonction proposée n'a pas la propriété de mise à jour. Il peut être difficile de trouver l'histoire nécessaire et suffisante à un traitement donné. D'autre part la définition formelle de cette caractéristique est basée sur la notion de bisimulation de systèmes de transitions, et la présentation de ces aspects dépasse le cadre des enseignements de DEUG.

Il est toutefois possible d'illustrer la notion de mémoire trop, ou pas assez riche, sur des exemples décrits par des automates (voir paragraphe 14.4 ci-dessous).

14.3.4 Relation avec la conception des systèmes réactifs

Le schéma général proposé permet de faire abstraction de la structure de l'algorithme itératif (progression, terminaison), et doit donc permettre de se concentrer sur le choix et la fonction de mise à jour des variables d'historique.

Lorsqu'on décrit un système réactif dans un langage synchrone, on s'abstrait également des aspects d'interaction avec un environnement extérieur, pour se concentrer sur le noyau réactif. Si l'on programme dans un style impératif, on raisonne sur la mémoire (explicitement en ARGOS, où il faut inventer les états, implicitement en ESTEREL, où les structures de contrôle du langage cachent des états). En LUSTRE, au contraire, tout se passe comme si on décrivait simplement la fonction de calcul des sorties (assimilable à AccumulationDe) et jamais l'histoire. C'est le compilateur qui détermine une histoire suffisante au traitement demandé. Dans le cas de la génération d'automate minimal, le compilateur calcule l'histoire *nécessaire et suffisante* au traitement demandé. En ARGOS, comme en ESTEREL, il est possible d'écrire des programmes non minimaux. Si le compilateur produit des systèmes de transitions étiquetées, un outil comme ALDÉBARAN permet de minimiser l'automate résultat vis-à-vis d'une relation d'équivalence comme la bisimulation.

14.4 Automate d'états finis pour la représentation des algorithmes

Dans le cas où l'histoire est un type à domaine de valeurs fini, ses valeurs peuvent être représentées par les états d'un automate, et les mises à jour par les transitions de cet automate. Nous proposons d'étudier les algorithmes itératifs en se ramenant à l'étude des automates.

14.4.1 Exemple de système interactif: l'interface homme/machine d'un éditeur

Dans cette étude de cas, on raisonne sur l'automate qui décrit les opérations du menu *FICHIER* d'un éditeur. Le travail consiste à s'interroger sur la mémoire nécessaire à la réalisation des opérations courantes. Par exemple, si, lorsque l'on quitte l'éditeur, le logiciel propose de sauvegarder les fichiers pour lesquels cette précaution n'a pas été prise, mais seulement ceux-là, c'est qu'il est capable de "se rappeler" qu'il y a eu une modification du tampon d'édition depuis la dernière sauvegarde. Cela demande simplement un élément booléen dans la mémoire complète.

La spécification informelle du comportement de l'éditeur conduit ainsi à se poser 3 questions :

- Le tampon est-il vide ?
- Existe-t-il un nom de fichier courant ?
- Y a-t-il eu une sauvegarde du tampon depuis la dernière modification ?

Répondre à ces trois questions suffit, en toute circonstance, à déterminer la réaction de l'éditeur à un événement extérieur de l'ensemble { sauver, sauver_dans, charger, modifier, fermer, quitter }.

A première vue, cela pourrait donc donner $2 \times 2 \times 2$ états pertinents. Toutefois certains d'entre eux sont inaccessibles. On développe le graphe d'états, à partir de l'état initial *tampon vide - pas de nom de fichier courant - tampon sauvegardé*, en envisageant les réactions à tous les événements extérieurs. 4 états seulement, parmi les 8 possibles, sont accessibles.

Le travail demandé est de deux ordres : travailler directement sur l'automate, pour prendre en compte des variantes de la spécification ; produire un algorithme itératif de simulation du comportement de l'éditeur, par un codage systématique des transitions de l'automate.

Parmi les variantes de spécification, on propose de supprimer l'action de fermeture de fichier. C'est un cas où la suppression de certaines transitions conduit à un automate non minimal pour la bisimulation. On pose alors la question : *tous les états sont-ils toujours nécessaires ?* On peut même coder l'automate dans un format accepté par ALDÉBARAN, et montrer aux étudiants le résultat de la réduction.

14.4.2 Preuve d'algorithmes représentés par des automates

On poursuit l'idée de représentation des algorithmes par des automates dans la direction de la preuve. Dans quelle mesure la représentation sous forme d'automate peut-elle aider à raisonner sur les algorithmes itératifs, et par exemple à prouver l'équivalence de deux algorithmes ? L'exemple exposé pourra sembler trop bien choisi pour ce propos, mais la démarche est applicable à tous les algorithmes de parcours de séquence, pour peu qu'ils soient à mémoire finie.

On s'intéresse au problème de la reconnaissance des multiples de 3 notés en base 2. La suite des chiffres est accessible grâce à une machine séquentielle qui fournit les chiffres des poids forts vers les poids faibles.

On écrit donc :

```
V : un entier ≥ 0
Démarrer ; V ⟵ 0
tant que non FinDeSéq
   V ⟵ 2 * V + EC                    { Calcul de la valeur entiere du nombre }
   Avancer
Ecrire (V mod 3 = 0)
```

Toutefois il est inutile de calculer la valeur entière du nombre, puisqu'on ne s'intéresse qu'à sa valeur modulo 3, qu'on peut calculer de manière incrémentale en utilisant la remarque suivante : ajouter un chiffre 1 consiste à multiplier par 2 puis à ajouter 1, ajouter un chiffre 0 consiste à multiplier par 2. Or, si $x \equiv 0[3]$, alors $2x \equiv 0[3]$ et $2x + 1 \equiv 1[3]$; si $x \equiv 1[3]$, alors $2x \equiv 2[3]$ et $2x + 1 \equiv 0[3]$; si $x \equiv 2[3]$, alors $2x \equiv 1[3]$ et $2x + 1 \equiv 2[3]$.

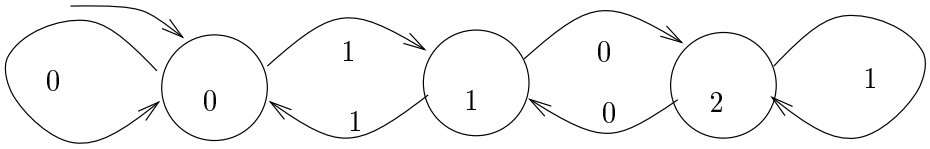


FIG. 14.3 – *Reconnaissance des multiples de 3 en base 2 – définition directe*

D'où l'algorithme :

```
V3 : un entier dans [0..2]
Démarrer ; V3 ⟵ 0
tant que non FinDeSéq
    selon V3 :
        V3 = 0 : si EC = 1 alors V3 ⟵ 1
        V3 = 1 : si EC = 0 alors V3 ⟵ 2 sinon V3 ⟵ 0
        V3 = 2 : si EC = 0 alors V3 ⟵ 1
    Avancer
Ecrire (V3 = 0)
```

Ce qui correspond à l'automate de la figure 14.3, où l'état 0 est accepteur.

D'autre part un théorème d'arithmétique établit qu'un nombre est congru, modulo $b+1$, à la différence entre la somme de ses chiffres en base b de rang pair, et la somme de ses chiffres en base b de rang impair.

La preuve est simple : on considère un entier c dont on suppose, sans perte de généralité, qu'il a un nombre pair de chiffres en base b (le cas échéant,on rajoute un chiffre 0 à gauche).

$$c = \sum_{i=0}^{n} c_i.b^i \text{ avec } n = 2m+1$$
$$c = \sum_{i=0}^{m} (c_{2i}.b^{2i} + c_{2i+1}.b^{2i+1}) = \sum_{i=0}^{m} (c_{2i} - c_{2i+1} + c_2 i.(b^{2i}-1) + c_{2i+1}.(b^{2i+1}+1))$$

or $b^{2i}-1$ et $b^{2i+1}+1$ sont des multiples de $b+1$, puisque -1 en est une racine, et ceci quel que soit $i \geq 0$. D'où

$$c \equiv \sum_{i=0}^{m} c_{2i} - \sum_{i=0}^{m} c_{2i+1} \quad [b+1]$$

Donc *les multiples de $b+1$ en base b sont tels que la différence entre la somme des chiffres de rang pair et la somme des chiffres de rang impair est aussi un multiple de $b+1$*. On écrit un algorithme basé sur cette propriété :

```
Sp, Si : des entiers ≥ 0 ; Pair : un booléen
Démarrer ; Sp ⟵ 0 ; Si ⟵ 0 ; Pair ⟵ faux
tant que non FinDeSéq
    si Pair alors Sp ⟵ Sp + EC sinon Si ⟵ Si + EC
    Pair ⟵ non Pair
    Avancer
Ecrire ((Sp − Si) mod 3 = 0)
```

Bien sûr la différence des sommes peut être calculée directement, et l'on obtient :

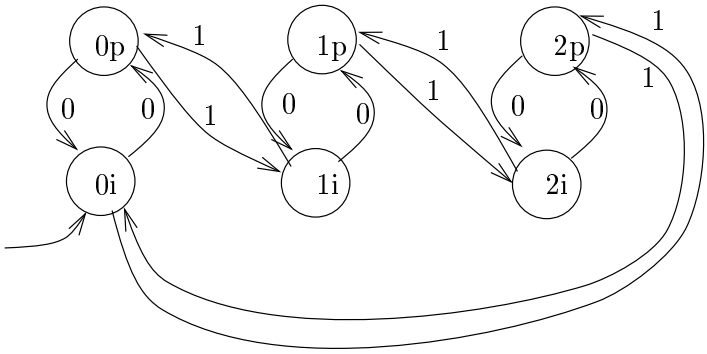


FIG. 14.4 – *Reconnaissance des multiples de 3 en base 2 – propriété*

```
S : un entier ; Pair : un booléen
Démarrer ; S ⟵ 0 ; Pair ⟵ faux
tant que non FinDeSéq
    si Pair alors S ⟵ S + EC sinon S ⟵ S − EC
    Avancer
Ecrire (S mod 3 = 0)
```

Et, de nouveau, il est inutile de calculer S puisque l'on ne s'intéresse qu'à sa valeur modulo 3, qui peut être déterminée incrémentalement. Le nouvel algorithme se déduit des propriétés du modulo exposées précédemment et de l'alternance entre rangs pairs et rangs impairs. Il correspond à l'automate de la figure 14.4, où les états 0i et 0p sont accepteurs.

Prouver l'équivalence de deux algorithmes (par exemple, prouver qu'ils conduisent à la même valuation des variables, s'ils travaillent sur les mêmes données) n'est pas chose aisée ; c'est même en général indécidable. Toutefois, dans le cas où les algorithmes de parcours séquentiel manipulent des histoires à domaine fini, on peut les représenter par des automates, et l'équivalence d'algorithmes se ramène alors à l'équivalence de traces de ces automates.

Ainsi il est aisé de montrer que les automates des figures 14.3 et 14.4 sont équivalents par l'équivalence de traces, et reconnaissent donc le même langage sur le vocabulaire $\{0, 1\}$ des chiffres en base 2.

Ceci fournit d'ailleurs une démonstration automatique du théorème d'arithmétique, pour une base particulière. On pourrait donner une définition générale des deux automates exhibés ci-dessus, pour une base quelconque, et montrer qu'ils sont équivalents pour l'équivalence de trace, mais ce ne serait plus automatique.

14.5 Proposition : description des algorithmes itératifs en ARGOS

Jusque là nous avons décrit les algorithmes itératifs par des automates, c'est-à-dire des structures plates. Je propose ici d'utiliser les constructions d'ARGOS comme la mise en parallèle et le raffinement, pour décrire des algorithmes dans lesquels une boucle unique impose de trop fortes contraintes.

La figure 14.5 donne un algorithme de calcul de la longueur moyenne des mots dans un texte

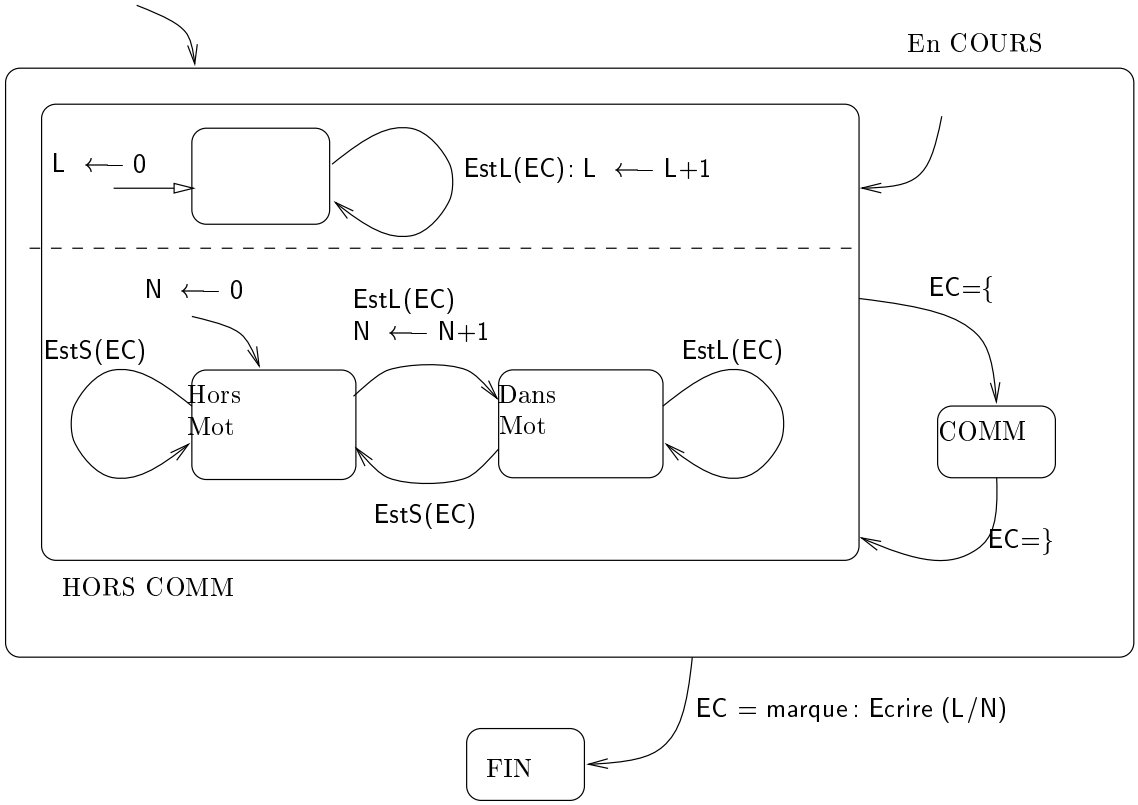


FIG. 14.5 – *Calcul de la longueur moyenne des mots dans un texte pourvu de commentaires*

pourvu de commentaires délimités aux deux extrémités par des accolades. Remarquons tout d'abord que la longueur moyenne des mots est simplement le résultat de la division du nombre de lettres par le nombre de mots (on oublie volontairement le cas où le nombre de lettres est nul). Il faut pour compter les lettres et les mots ne tenir compte que des parties *significatives* du texte, c'est-à-dire celles situées hors des commentaires.

Le programme ARGOS reflète exactement la structure qu'il faut plaquer sur le texte lu pour réaliser les calculs demandés : la structure principale exprime que le dernier caractère (la marque de fin) est traité à part, et que tout le comportement qui raffine l'état EN COURS doit être interrompu lorsqu'on lit la marque. La structure intérieure a deux états COMM et HORS COMM qui définissent les portions de commentaires : elles commencent par une accolade ouvrante, et se terminent par une accolade fermante. Au début du texte, on est hors de tout commentaire. Enfin la structure qui raffine l'état HORS COMM est une composition parallèle : le premier composant s'occupe de compter les lettres, et n'a pas de mémoire ; le deuxième composant définit la notion de mot : on reconnaît un début de mot comme un couple constitué d'un séparateur et d'une lettre : ceci nécessite une mémoire booléenne, l'état Hors Mot signifiant "*le dernier caractère lu était un séparateur*". Notons que le début du texte est aussi un début de mot.

Quelques remarques :

– Si l'on écrit directement un algorithme itératif de parcours de la séquence, basé sur cette structure du texte, on obtient assez naturellement des itérations emboîtées :

```
Démarrer
Tant que non FinDeSéq
   tant que non FinDeSéq et puis EstS(EC) : Avancer
   ...
```

Les itérations emboîtées sont également conditionnées par non FinDeSéq. On observe souvent chez les étudiants des écritures incorrectes basées sur une interprétation erronée de la sémantique du tant que, qui consiste à croire que la condition du tant que le plus externe "*s'ajoute*", en quelque sorte, aux conditions des itérations emboîtées, et qu'il est donc inutile de la rappeler.

Nous écrivions au chapitre 9 que *Dans les langages séquentiels, il est possible de comprendre le corps d'une itération en s'abstrayant de la condition de continuation. En revanche, un chien de garde* ESTEREL *interagit avec son corps* à tout instant *de l'exécution de celui-ci.* Cette même remarque s'applique ici : lorsque l'algorithme est décrit en ARGOS, l'automate contrôleur du raffinement (qui joue le rôle du chien de garde) fonctionne en parallèle avec le processus raffinant : lui seul s'occupe de la détection de fin de séquence, et contrôle tous les parcours décrits dans les processus raffinants.

– La composition parallèle (sans communication ici) correspond très exactement au mécanisme de repliage des fonctions. Si l'on écrit d'une part un algorithme de parcours séquentiel qui compte les lettres, d'autre part un algorithme de parcours séquentiel qui compte les mots, il est ensuite aisé de les rassembler pour décrire le même calcul en un seul parcours de la séquence. En style fonctionnel, cela revient à fusionner deux fonctions à un résultat en une seule fonction à résultat couple.

  La composition parallèle permet de décrire l'indépendance des deux calculs, sans imposer la structure de mots à la portion de programme dans laquelle on compte les lettres.

14.6 Conclusion provisoire

Le point de vue développé ici sur la construction des itérations conduit à un raisonnement sur la quantité et la nature des *informations* nécessaires à un traitement donné, qui peut être développé indépendamment de toute technique de programmation. En un sens, et au risque de paraître pompeux, disons que c'est "l'essence" du raisonnement informatique.

Ce niveau de réflexion peut sembler hors de portée des étudiants de DEUG. La manipulation *éditeur de textes* permet de concrétiser quelque peu les choses. L'expérience montre que certains étudiants produisent intuitivement la réduction d'un automate par l'équivalence de trace.

Chapitre 15

Description de circuits en Lustre et VHDL, niveau Licence

Sources : *Ce chapitre rassemble quelques réflexions issues du travail dans l'équipe du module* ALM *(Architectures Logicielles et Matérielles) de première année de l'Ecole Universitaire d'Informatique de Grenoble. Les enseignements décrits sont anciens et il est parfois difficile d'en retrouver toutes les sources, mais pour les 6 dernières années il faut remercier P. Amblard, J.-C. Fernandez, F. Lagnier, P. Sicard et Ph. Waille.*

15.1 Le contexte

Le module *Architectures Logicielles et Matérielles* rassemble dans une présentation unifiée les enseignements traditionnels de logiciel de base (programmation en assembleur, vie des programmes, première approche des systèmes d'exploitation) et d'architecture des machines (algèbre de Boole, circuits combinatoires et séquentiels, structure des processeurs, etc.). Pour les aspects matériels, l'objectif est de faire comprendre la structure des circuits et de montrer, en particulier, que la démarche de conception de ces "algorithmes matériels" est identique à la démarche de conception des algorithmes "logiciels" (spécification de blocs, abstraction et réutilisation...). On arrive ainsi à la conclusion que de nombreux problèmes peuvent recevoir des solutions logicielles ou matérielles, voire mixtes, selon les critères que l'on privilégie.

Cette année les travaux pratiques de conception de circuits sont réalisés en VHDL. On utilise des outils commerciaux de simulation et un outil de synthèse pour carte Xilinx. Puisque l'enseignement vise à faire comprendre dans les détails la structure des circuits, et la frontière entre matériel et logiciel, il est hors de question d'utiliser le langage VHDL dans toute sa puissance d'expression, et de faire confiance aux outils de synthèse pour produire des circuits difficilement analysables. On se restreint donc à un sous-ensemble dit *structurel*, qui ressemble énormément à LUSTRE booléen avec tableaux (on ne manipule pas directement les entiers, par exemple, mais des nappes de fils). Les mêmes sujets de travaux pratiques sont traités en VHDL pour une partie de la promotion de première année, et en LUSTRE pour l'autre partie (avec utilisation d'outils de Vérilog).

Puisque la distance est si faible entre LUSTRE et le sous-ensemble de VHDL utilisé, l'idée de traduire l'un dans l'autre est assez naturelle. Si l'on décrit un circuit en LUSTRE, on bénéficie de tous les environnements de preuve, test et simulation de la communauté synchrone. Une traduction en VHDL permet ensuite d'accéder aux outils de synthèse pour les cartes Xilinx

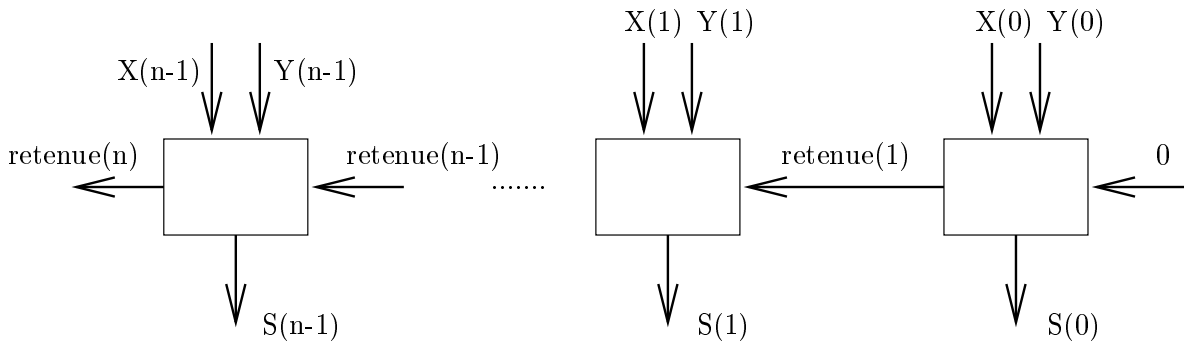


FIG. 15.1 – *Structure d'un additionneur n bits*

(rappelons que la thèse de F. Rocheteau [Roc92] traitait de la programmation des cartes Xilinx en LUSTRE avec tableaux).

D'autre part, puisqu'on sait mélanger ARGOS et LUSTRE, il ne doit pas être trop difficile de mélanger ARGOS et VHDL. Les techniques de réalisation matérielle d'algorithmes dits *partie contrôle/partie opérative* constituent d'excellents exemples de description mixte impératif/déclaratif.

15.2 Traduction Lustre vers VHDL

Nous donnons ici un exemple de circuit décrit en LUSTRE et en VHDL (tables 15.1 et 15.2). La version LUSTRE est due à J.-C. Fernandez et la version VHDL à un étudiant. C'est un additionneur n bits décrit comme composition de n additionneurs 1 bit (voir figure 15.1). Les deux programmes sont très similaires, à des détails de syntaxe près. La principale difficulté de traduction de l'un vers l'autre réside dans les mécanismes de description des circuits génériques répétitifs. En LUSTRE on dispose d'opérations sur des tranches de tableaux. En VHDL on écrit des boucles FOR et on manipule explicitement les indices dans les tableaux.

15.3 Argos et VHDL

Les outils VHDL dont on dispose acceptent des descriptions de parties contrôle sous forme de machines de Moore ou de Mealy. Dans les travaux pratiques, nous montrons comment coder systématiquement, à la main, une machine de Moore en équations booléennes.

L'utilisation d'ARGOS n'apporte donc rien pour la description de parties contrôles simples, données sous forme d'automates. L'intérêt d'ARGOS réside dans ses constructions, qui autorisent la description des parties contrôle par des coopérations d'automates, et non plus seulement par des automates plats. Cela peut permettre, en utilisant la structure du programme, de déterminer un codage astucieux et économique des états (voir [DY90] déjà cité au chapitre 5 et [VNG91] qui décrit le langage des SPECCHARTS, une variante des Statecharts compilable en VHDL).

Enfin on pourrait proposer une syntaxe pour ARGOS avec tableaux, en s'inspirant à la fois de LUSTRE avec tableaux et de la syntaxe de VHDL pour les vecteurs.

```
node add1   ( x, y, cin : bool)
    returns ( s, cout : bool);
let
    s = x xor y xor cin;
    cout = if cin then (x or y) else (x and y);
tel

node addn    ( const n : int; X, Y : bool^n)
     returns ( S : bool^n; cout, v : bool);
var
    C : bool^n;
let
    (S[0], C[0]) = add1bit(X[0], Y[0], false);
    (S[1..n-1], C[1..n-1]) =
        add1bit(X[1..n-1],Y[1..n-1],C[0..n-2]);
    cout = C[n-1];
    v = (X[n-1] and Y[n-1] and not S[n-1]) or
    (not X[n-1] and not Y[n-1] and S[n-1]);
tel

node add4 ( X, Y : bool^4)
     returns ( S : bool^4; cout, v : bool);
let
    S, cout, v = addn (4, X, Y);
tel
```

TAB. 15.1 – *Additionneur n bits en* LUSTRE

```
entity add1 is port (x, y, cin : in bit;  s, cout : out bit);
end add1;
architecture add1_int of add1 is
begin
   s <= x xor y xor cin;
   cout <= (x and y) or (x and cin) or (y and cin);
end add1_int;

ENTITY addn IS
        GENERIC (n:INTEGER);
        PORT ( X, Y : IN  BIT_VECTOR (n-1 downto 0);
               S : OUT BIT_VECTOR (n-1 downto 0); cout, v  : OUT BIT );
END addn;

ARCHITECTURE addn_int OF addn IS
        COMPONENT add1 PORT (x,y,cin : IN BIT ; s, cout : OUT BIT );
        END COMPONENT;
        FOR ALL : add1 USE ENTITY work.add1(add1_int);

        SIGNAL retenue : BIT_VECTOR (n downto 0);
BEGIN
           retenue (0) <= '0';
           cout <= retenue (n);
           v <= (X(n-1) and Y(n-1) and not S(n-1)) or
                (not X(n-1) and not Y(n-1) and S(n-1)) ;
           etiq1 : FOR i IN 0 TO n-1 GENERATE
           etiq2 : add1 PORT MAP (X(i),Y(i),retenue(i),S(i),retenue(i+1));
           END GENERATE;
END addn_int;

entity add4 (X4, Y4 : IN  BIT_VECTOR (3 downto 0);
             S4 : OUT BIT_VECTOR (3 downto 0); cout4, v4  : OUT BIT );
END add4 ;

ARCHITECTURE add4_int OF   add4  IS
    CONSTANT NBits : INTEGER := 4 ;
    COMPONENT addn
        GENERIC (n : INTEGER) ;
        PORT    ( X, Y : IN  BIT_VECTOR (n-1 downto 0);
                  S : OUT BIT_VECTOR (n-1 downto 0); cout, v  : OUT BIT );
    END COMPONENT;
    FOR ALL addn USE ENTITY work.addn(addn_int);
BEGIN
        etiq: addn     GENERIC MAP (n => NBits)
                       PORT MAP (X4, Y4, S4, cout4, v4) ;
END add4_int ;
```

TAB. 15.2 – *Additionneur n bits en VHDL structurel*

Chapitre 16

Quelques remarques en guise de conclusion

On trouvera les points "bilan" et "perspectives" d'une conclusion classique dans les remarques présentées ci-dessous. Le résumé des activités fait partie de l'introduction. Les remarques se rangent dans un ou plusieurs points ci-dessous :

- Tout d'abord, une tentative de réponse à la question : "*à quoi ça sert ?*", qu'il est assez naturel de se poser après 10 ans de travail sur un sujet — et qu'on me posera sans doute.
- L'exposé d'une direction pour la suite, à la fois direction de recherche et choix de la nature du travail. Cela s'appuie naturellement sur un bilan des résultats obtenus et de leur diffusion, mais aussi sur un ensemble de réflexions plus personnelles à propos de la place d'un enseignant/chercheur dans une équipe de recherche engagée dans des collaborations industrielles.
- Un retour sur les influences réciproques entre enseignement et recherche, accompagné de quelques réflexions sur la place de mon domaine de recherche au sein de la discipline informatique. En une dizaine d'années d'enseignement de sujets assez divers à des publics variés, on finit par se forger une certaine image globale de la discipline, qui rejaillit sur la manière d'aborder les problèmes, sur la manière de les enseigner et, de façon plus fondamentale, sur le choix de ce que l'on enseigne. Il s'agit ici de mettre en forme cette image globale — aussi subjective soit-elle — pour essayer de répondre à la question : "*Quels résultats/approches/techniques de ce domaine de recherche sont ou pourraient être utilisés en enseignement, pourquoi, et à quel niveau ?*".

Recherche

Bilan

Puisque le titre de ce document annonce "un *langage* à base d'automates", la question se pose de savoir si l'on peut considérer ARGOS comme un véritable langage. La réponse est non, pour une grande variété de raisons.

En particulier, je ne crois pas à un véritable langage dont la syntaxe serait uniquement graphique. Les défauts d'une syntaxe graphique sont exposés en introduction du chapitre 9 comme l'une des motivations de l'approche ARGOS+ESTEREL, et les difficultés liées à la définition

d'un outil d'édition sont relatées au chapitre 12 (la notion d'analyseur syntaxique d'un langage à deux dimensions n'est pas aussi naturelle que ne l'est devenue celle de l'analyse syntaxique de texte, et les éditeurs sont donc nécessairement syntaxiques). D'autre part, indépendamment de la syntaxe concrète, qu'on pourrait après tout choisir textuelle, les constructeurs d'ARGOS sont en nombre limité, et il est assez contraignant d'utiliser des automates pour décrire certains types de comportements réactifs. Enfin l'environnement de programmation autour d'ARGOS n'est qu'un ensemble de maquettes plus ou moins éprouvées, complété par des prototypes de connexion à différents outils du domaine.

Toutefois cet état de faits relève d'une décision délibérée. Les travaux sur ARGOS n'ont pas été réalisés avec un objectif de transfert rapide vers l'industrie, ni même de large diffusion. L'industrialisation de LUSTRE ou ESTEREL montre bien l'ampleur de la tâche, et la nature des activités qu'il faut avoir : développement et maintenance de logiciel très conséquent, études de cas nombreuses, etc. Ces activités diverses et difficiles — car il y a un pas énorme entre le développement d'une petite maquette à des fins d'expérimentation dans un environnement de recherche, et le développement d'un prototype au moins maintenable et si possible évolutif — ne peuvent être menées qu'en équipe et, à mon sens, par du personnel "suffisamment" permanent pour assurer un minimum de continuité. Pour se lancer dans des activités de développement avec des objectifs de diffusion ou de transfert, il faut donc s'en donner les moyens matériels et humains, et cela n'est envisageable qu'une fois le sujet raisonnablement cerné.

L'étude du langage ARGOS proprement dit a joué jusqu'à présent un rôle exploratoire, et la question de son transfert ne se pose véritablement que maintenant. En effet, les problèmes de fond d'un langage synchrone à base d'automates, et d'un environnement de programmation synchrone multi-langages, ont été étudiés et résolus.

Malgré l'absence d'un prototype largement diffusable, les idées d'ARGOS émergent, parmi les douzaines de sémantiques proposées pour les Statecharts, comme la base formelle raisonnable d'un langage réactif offrant des automates hiérarchiques. Joe Buck, de Synopsys, travaille sur une extension d'ESTEREL qui introduit une construction tirée du raffinement d'ARGOS. L'équipe d'Axel Poigné à la GMD travaille sur un environnement de programmation synchrone intégrant LUSTRE, ESTEREL et ARGOS. Charles André, à Nice, a défini le formalisme des SyncCharts, où la structure hiérarchique des automates a la sémantique du raffinement Argos ; les SyncCharts sont en voie d'industrialisation. Nous travaillons d'autre part à l'introduction de structures à états et transitions dans SAO+ (l'environnement de programmation basé sur Lustre et industrialisé par la société Vérilog), et la sémantique d'Argos est déjà une base de travail intéressante.

D'autre part le prototype de compilateur, connecté à l'outil KRONOS de Vérimag, a été utilisé avec succès dans le cadre du stage post-doctoral de Muriel Jourdan dans l'équipe de Bernard Espiau.

En ce qui concerne les travaux sur le multi-langages, étudier la programmation mixte ARGOS/LUSTRE ou ARGOS/ESTEREL a permis d'identifier ce que peut apporter ARGOS à un environnement de programmation des systèmes réactifs basé sur l'un des langages bien établis. Ainsi le langage mixte ARGOS/ESTEREL peut-être vu comme une proposition d'extension d'ESTEREL, par introduction d'une structure de contrôle temporelle inspirée du raffinement ARGOS, mais héritant de la richesse des modes de terminaison ESTEREL. Le langage mixte ARGOS/LUSTRE permet de s'intéresser au problème de la réinitialisation dans un langage flot de données, et d'étudier comment les constructions d'automates se marient avec une structure flot de données.

En résumé, les travaux autour d'ARGOS ont permis d'explorer les compositions d'automates synchrones, et de montrer les points de convergence entre les langages de description graphiques

à la Statecharts et un langage parfaitement défini comme ESTEREL ou LUSTRE. Les extensions temporelles et hybrides, ainsi que l'utilisation des automates en algorithmique, suggèrent d'autres champs d'exploration.

Perspectives

Les divers travaux autour de la définition d'ARGOS me semblent maintenant parvenus à un point de convergence.

Pour les raisons énoncées plus haut, le développement complet de ce qui serait un véritable langage, muni de son environnement de programmation propre, ne constitue pas nécessairement la seule poursuite envisageable. Le travail me semble pouvoir maintenant être organisé selon deux axes :

- L'étude des compositions d'automates dans des contextes plus larges que le contexte des langages synchrones, avec de nouveaux objectifs d'exploration de la définition des langages, plutôt que de production d'environnements de programmation. Ceci demande le développement d'un prototype de compilateur ARGOS comme moteur de composition d'automates synchrones décorés, utilisable comme langage de description dans tout environnement qui manipule des automates plats. C'est ce que nous avons fait pour POLKA, et l'idée est généralisable. Un tel environnement permettrait également d'expérimenter les idées sur l'introduction du non déterminisme dans les langages synchrones, et l'utilisation des compositions synchrones d'automates dans l'enseignement de l'algorithmique. C'est un projet maîtrisable en développement logiciel, dans un contexte universitaire.

- L'utilisation et l'application de l'expérience et des résultats acquis sur les constructions d'automates dans le cadre synchrone.

  La demande industrielle nous oriente vers la définition d'une extension de LUSTRE à l'aide d'automates, avec un point de vue assez différent de l'approche "édition de liens" que nous suivions jusqu'ici. Il s'agit d'utiliser un automate comme structure de contrôle appliquée à la définition d'un ensemble de flots X_1, X_2, etc., dont chaque état définit *un* mode d'évolution possible de chacun des flots. La définition globale des flots est une *union* de ces différentes définitions. Un exemple typique consiste à calculer une fonction affine par morceaux : dans un état le flot X est défini par `X = pre(X) + 1`, dans un autre état par `X = pre(X) - 1`. Il s'agit bien du même X. On peut proposer une syntaxe où des équations LUSTRE apparaissent dans les états d'un automate, mais l'inteprétation n'est plus celle du raffinement ARGOS: l'ensemble d'équations d'un état ne constitue pas un processus qu'il faudrait détruire et relancer dans son état initial. Si l'on adopte une vision circuit séquentiel, le changement d'état consiste à remplacer, en cours de fonctionnement, une fonction de transition par une autre, en conservant la mémoire. La variable X est ainsi *globale* à l'automate.

  Cette approche, si elle semble ne poser que des problèmes théoriques déjà bien cernés, pose des problèmes spécifiquement "langage". Il sera toujours possible de proposer une sémantique pour un langage mixte de ce genre, mais correspondra-t-elle à l'interprétation intuitive ? Si la définition d'une variable X apparaît dans certains états et non dans d'autres, doit-on considérer qu'il existe une définition par défaut `X = pre(X)`, ou bien que les états jouent le rôle d'horloges LUSTRE, et que la variable X est non disponible lorsque non explicitement définie ? La structure d'automate plat suffit-elle, ou doit-on prévoir l'extension

à des contrôleurs structurés, au moins hiérarchiquement ? Toutes ces questions doivent être étudiées à partir d'un ensemble d'exemples significatifs.

D'autre part, dans un contexte plus théorique, et comme préalable à l'introduction de non déterminisme dans les constructions synchrones, il faut se pencher sur l'étude d'une relation d'équivalence d'objets non déterministes, compatible avec le point de vue selon lequel un objet non déterministe représente un ensemble d'objets déterministes. C'est un travail peut-être moins technique que de bibliographie, puisque ce point de vue prévaut dans une certaine communauté, à propos de raffinements de spécifications vers des programmes. Mais il me semble intéressant d'essayer de comprendre les rapprochements possibles entre les résultats de cette communauté, et ceux de la communauté algèbres de processus — que je connais mieux — où la bisimulation est reine.

Enseignement et recherche

La position d'enseignant/chercheur dans une UFR qui a la charge des enseignements d'informatique du premier jusqu'au troisième cycle offre de nombreuses possibilités d'intervention.

Enseigner directement son sujet de recherche est une possibilité, mais c'est par nature limité à des enseignements optionnels d'ouverture ou à des enseignements de troisième cycle. D'autre part c'est une vision à mon avis un peu étroite de ce que peut apporter un domaine de recherche à l'enseignement ; il y a bien des choses à enseigner aux étudiants de premier et second cycle, avant d'en faire des experts en systèmes réactifs (par exemple). Une autre façon de voir les choses, à mon avis plus riche, est de considérer que chaque domaine se caractérise par des préoccupations, une démarche, des résultats et des techniques particulières, qu'il est intéressant d'utiliser pour colorer l'enseignement général en premier et second cycle. La richesse des équipes d'enseignants vient souvent — pour ne pas dire toujours — d'une confrontation de points de vue ; et ces points de vue sont souvent le reflet des préoccupations d'environnements de recherche différents.

Utilisation de la culture "méthodes formelles pour les systèmes réactifs"

Plus précisément, quelles sont les préoccupations, la démarche, les résultats et les techniques utilisés dans la conception et la validation des systèmes réactifs ? La *calculabilité* est à la base de nombreux choix, puisqu'on essaie toujours de trouver des compromis entre décidabilité et expressivité des modèles de programmes. L'analyse de *complexité* est une priorité parce que les algorithmes sont étudiés pour être implantés et traiter des exemples conséquents ; de plus, c'est un domaine où l'on fait peu de distinction, en pratique, entre des problèmes de très grande complexité (qu'on ne pourrait résoudre qu'avec des algorithmes très coûteux) et des problèmes indécidables. D'où l'émergence des techniques de *vérification approchée*, qui conduisent à réfléchir sur les abstractions raisonnables des programmes. La théorie du point fixe et la *sémantique* sont nécessaires puisqu'on s'intéresse à la définition formelle des langages ; la *compilation* et les techniques usuelles de développement des compilateurs sont indispensables dans un cadre qui privilégie l'interconnexion des outils par une grande variété de formats et de langages intermédiaires.

Sans prétendre avoir des compétences très élaborées sur chacun de ces différents points, il me semble que le travail de définition de méthodes et outils de validation des systèmes réactifs, sachant que le preuve formelle du système réel est hors d'atteinte, et en tenant compte de

préoccupations relatives à la définition de langages, donne une expérience assez large, depuis les aspects formels jusqu'à l'implantation.

Cette expérience et ce point de vue m'ont confirmé l'idée que les enseignements de base, disons de première année de second cycle pour fixer les idées, sont intimement liés, et gagneraient à être présentés de manière moins indépendante. Les outils théoriques ont par exemple leur place dans des enseignements du style "mathématiques pour l'informatique" [1], et une bonne partie des sujets qui y sont abordés est également à la base de toutes les définitions syntaxiques et sémantiques des langages. Parallèlement, l'expérience sur la définition et l'implantation des langages s'acquiert avec l'algorithmique en général, la compilation, des aspects de plus bas niveau comme le logiciel de base, et même la conception de circuits décrits dans des langages comme VHDL [2]. La mise en place d'un projet de programmation d'un assembleur/éditeur de liens pour sparc, sous la forme d'une collaboration entre les modules L&P et ALM, m'a convaincue de l'intérêt, pour les étudiants, d'un contexte dans lequel les connaissances acquises dans deux modules différents, avec des ambiances d'enseignement différentes, sont utilisées conjointement. J'aimerais compléter le schéma en étendant le projet aux modules qui traitent des outils théoriques nécessaires, dont par exemple les machines d'états finies utilisées en analyse lexicale.

Finalement, du domaine plus particulier des *langages synchrones* pour la description et la validation des systèmes réactifs, on peut extraire la notion de *parallélisme de description* — par opposition au parallélisme d'*exécution*, qu'il soit réel ou simulé — et essayer de l'utiliser comme paradigme de programmation. Dans [SFLM93], on insiste sur la variété des modes d'expression, et on illustre ce discours par l'étude des paradigmes fonctionnel, actionnel et relationnel. Il faudrait compléter cet ensemble par la programmation concurrente ou logique, l'orienté objet, les tableaux, etc. Pour les aspects de programmation concurrente, il me semble que les langages synchrones sont le moyen idéal d'introduire la conception parallèle sans s'encombrer de vrai parallélisme — c'est-à-dire de machines parallèles et d'algorithmique distribuée — simplement comme un autre moyen de concevoir les programmes séquentiels. Nous avons vu au chapitre 14 paragraphe 14.5 une version ARGOS d'un algorithme de parcours de séquence de lettres qui calcule la longueur moyenne des mots. le parallélisme qui y apparaît est purement descriptif, et le programme est fait pour être compilé en code séquentiel. C'est la même notion de "parallélisme" que celle qui apparaît en filigrane dans la technique de *repliage* des fonctions : comment faire en un seul parcours d'une séquence, à la fois le calcul de la longueur et le calcul de l'élément maximum ? Il s'agit en fait de décrire l'exécution "parallèle" de deux algorithmes de parcours de séquence.

A propos du transfert des méthodes formelles

Une question rituelle se pose dans les environnements de recherche où l'on s'occupe de *"méthodes formelles"*, quelle que soit l'étendue du domaine que cela recouvre : *"comment assurer le transfert de ces fameuses méthodes formelles, de la recherche vers l'industrie ?"*. J'aimerais ici apporter quelques arguments pour une réponse à cette question, en adoptant le point de vue d'un enseignant-chercheur, pour qui le problème est aussi celui du transfert des méthodes formelles de la recherche vers la *formation*.

C'est une vision à long terme, qui certes demande de la patience dans la simple observation des résultats. Mais, quand on y songe un peu, c'est bien aujourd'hui qu'il faut se préoccuper de

1. à Grenoble il existe plusieurs modules intitulés "Outils Formels pour l'Informatique" au niveau Bac+3
2. à Grenoble, cela concerne donc à Bac+3 les modules L&P — Langages et programmation — et ALM — Architectures Logicielles et Matérielles

la formation des ingénieurs qui réaliseront les systèmes informatiques des avions dont dépendra notre sécurité dans 10 ou 15 ans.

La question du transfert des méthodes formelles recouvre donc aussi la question *"comment enseigner les méthodes formelles?"*. Commençons par préciser quelque peu les objectifs d'un tel enseignement, ainsi que les contenus susceptibles de remplir ces objectifs.

Une manière de répondre à ces questions — *comment, pourquoi et quoi?* — consiste à proposer, par exemple, l'étude de la théorie des logiques temporelles dans un cours de DEA. Une autre manière, qui suppose moins d'acquis du public, consiste à définir une option de niveau Bac+4 à propos des systèmes temps-réel, "spécification en logique temporelle et validation de protocoles décrits en Lotos" ; ou bien "des observateurs synchrones pour un système de contrôle/commande décrit en Lustre". Mais cela suppose encore de nombreux acquis.

A l'opposé, si l'on cherche à décomposer très finement l'ensemble des techniques et méthodes qu'il faudrait maîtriser pour assimiler parfaitement un cours sur la spécification formelle d'un système réactif, on se rend compte que nombre des points qui apparaissent peuvent être présentés, d'une part indépendamment des systèmes réactifs, et d'autre part assez tôt dans la formation, par exemple en premier cycle.

A mon sens, on commence à enseigner les méthodes formelles en habituant les étudiants à l'idée qu'un programme est, certes, exécutable, mais que c'est aussi un objet sur lequel il est possible de raisonner : on peut analyser un programme pour lui trouver des propriétés particulières ; le transformer de manière systématique en respectant le sens ; évaluer sa complexité en temps, etc. On a même le droit, toutes choses égales par ailleurs, de préférer une version à une autre pour des critères d'élégance et de simplicité. On oublie sans doute trop souvent que cette idée est loin d'être naturelle pour des étudiants de premier cycle qui se sont forgé leur image personnelle de la discipline dans la presse informatique.

La deuxième étape dans cette formation de longue haleine peut consister à pratiquer la preuve ou la comparaison de programmes dans des cas très simples, par exemple en étudiant des systèmes séquentiels à nombre fini d'états. Il est même possible, dans ces cas simples, de passer à l'étage supérieur : l'automatisation de telles preuves. je m'appuie ici sur l'expérience décrite au chapitre 14, où il s'agissait de programmer l'interface d'un éditeur de textes à l'aide d'automates. On arrivait assez naturellement, et sans besoin de développer une théorie élaborée, à la notion d'équivalence de trace de systèmes de transitions étiquetées, et donc à la notion d'équivalence des programmes correspondants, produits de manière systématique.

L'étape suivante consiste à travailler sur la preuve de programmes séquentiels non finis, en introduisant une représentation symbolique des états par des prédicats. On peut d'ailleurs assez vite se restreindre à l'utilisation de ces assertions pour la construction des programmes plutôt que pour leur preuve a posteriori.

Il reste finalement à particulariser cette culture de raisonnement plus ou moins automatisable sur les programmes, au cas des systèmes réactifs.

Bibliographie

[AB84] Austry (D.) et Boudol (G.). – Algèbre de processus et synchronisation. *TCS*, vol. 30, avril 1984.

[ABC94] Arnold (A.), Bégay (D.) et Crubillé (P.). – *Construction and analysis of transition systems with MEC.* – World Scientific Publishing, 1994.

[ACH+92] Alur (R.), Courcoubetis (C.), Halbwachs (N.), Dill (D.) et Wong-Toi (H.). – Minimization of timed transition systems (extended abstract). *In : CONCUR'92, Stony Brook.* – LNCS 630, Springer Verlag.

[AD90] Alur (R.) et Dill (D.). – Automata for modeling real-time systems. *In : Proceedings of ICALP 90*, éd. par Paterson (M.S.). pp. 322–335. – Springer Verlag.

[AG95] André (Charles) et Gaffé (Daniel). – *Sequential Function Charts: a Synchronous Point of View.* – Rapport technique n° RR 95–08, Sophia-Antipolis, France, I3S, février 1995.

[And96] André (C.). – Representation and analysis of reactive behaviors: A synchronous approach. *In : Computational Engineering in Systems Applications (CESA).* pp. 19–29. – Lille (F), juillet 1996.

[Arn92] Arnold (A.). – *Systèmes de transitions finis et sémantique des processus communicants.* – Masson, 1992.

[BCM90] Berthet (C.), Coudert (O.) et Madre (J. C.). – New ideas on symbolic manipulations of finite state machines. *In : International Conference on Computer Design (ICCD), Cambridge.*

[Ber91] Berry (G.). – A hardware implementation of pure ESTEREL. *In : ACM Workshop on Formal Methods in VLSI Design, Miami.*

[Ber93] Berry (G.). – Preemption in concurrent systems. *In : Proceedings of FSTTCS 93.* – Springer Verlag, LNCS 761.

[BG92] Berry (G.) et Gonthier (G.). – The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, vol. 19, n° 2, 1992, pp. 87–152.

[CCI89] CCITT. – *Specification and Description Language.* – Rapport technique n° Tome X, ITU, 1989.

[CGP94] Caspi (P.), Girault (A.) et Pilaud (D.). – Distributing reactive systems. *In : Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94.* – Las Vegas, USA, octobre 1994.

[CGS91] Courcoubetis (C.), Graf (S.) et Sifakis (J.). – An algebra of boolean processes. *In : Workshop on Computer-Aided Verification.* – Aalborg, juillet 1991.

[CHPP87] Caspi (P.), Halbwachs (N.), Pilaud (D.) et Plaice (J.). – LUSTRE, a declarative language for programming synchronous systems. *In : 14th Symposium on Principles of Programming Languages.* – Munich, janvier 1987.

[CS95] C2A-SYNCHRON. – *The common format of synchronous languages – The declarative code DC version 1.0.* – Technical report, SYNCHRON project, octobre 1995.

[CTC90] Costagliola (G.), Tomita (M.) et Chang (S.). – Dr parsers: a generalization of lr parsers. *In : IEEE Workshop on Visual Languages.* pp. 174–180. – Skokie (Illinois), octobre 1990.

[CTC91] Costagliola (G.), Tomita (M.) et Chang (S.). – A generalized parser for 2-d languages. *In : IEEE Workshop on Visual Languages.* pp. 98–104. – Kobe (Japan), octobre 1991.

[CV89] Collette (P.) et Villar (B.). – *Intégration de formalismes déclaratifs et impératifs pour la spécification des systèmes réactifs.* – Thèse, Universié de Louvain-La-Neuve, 1989.

[DY90] Drusinsky-Yoresh (D.). – A simple single-block implementation and efficient state-assignments for statecharts. *In : Synthesis and Simulation Meeting and International Interchange (SASIMI).* – Kyoto, 1990.

[GBBG85] Guernic (P. Le), Benveniste (A.), Bournai (P.) et Gauthier (T.). – *Signal: A Data Flow Oriented Language for Signal Processing.* – Rapport technique, IRISA report 246, IRISA, Rennes, France, 1985.

[GC92] Girault (A.) et Caspi (P.). – An algorithm for distributing a finite transition system on a shared/distributed memory system. *In : PARLE'92, Paris.*

[Hal] Halbwachs (N.). – ...

[Hal93] Halbwachs (N.). – Delay analysis in synchronous programs. *In : Fifth Conference on Computer-Aided Verification.* – Elounda (Greece), juillet 1993.

[Har87] Harel (D.). – Statecharts : A visual approach to complex systems. *Science of Computer Programming*, vol. 8, 1987, pp. 231–275.

[HGdR88] Huizing (C.), Gerth (R.) et de Roever (W.P.). – Modelling statecharts behaviour in a fully abstract way. *In :* 13^{th} *CAAP.* – LNCS 299, Springer Verlag.

[HLR93] Halbwachs (N.), Lagnier (F.) et Raymond (P.). – Synchronous observers and the verification of reactive systems. *In : Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, éd. par Nivat (M.), Rattray (C.), Rus (T.) et Scollo (G.). – Twente, juin 1993.

[HM95] Halbwachs (N.) et Maraninchi (F.). – On the symbolic analysis of combinational loops in circuits and synchronous programs. *In : EUROMICRO.* – Como, Italy, septembre 1995.

[HMP95] Halbwachs (N.), Maraninchi (F.) et Proy (Y. E.). – The railroad crossing problem, modeling with hybrid argos - analysis with polka. *In : Second European Workshop on Real-Time and Hybrid Systems.* – Grenoble (France), juin 1995.

[HNSY92] Henzinger (T.), Nicollin (X.), Sifakis (J.) et Yovine (S.). – Symbolic model-checking for real-time systems. *In : LICS'92.*

[HP88] Harel (D.) et Pnueli (A.). – On the development of reactive systems. *In : Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* – NATO ASI series F, Springer Verlag.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, vol. 79, n° 9, septembre 1991.

[JLMR93] Jourdan (M.), Lagnier (F.), Maraninchi (F.) et Raymond (P.). – Embedding declarative subprograms into imperative constructs. *In : Fifth International Symposium on Programming Language Implementation and Logic Programming,* Tallin, Estonia. – Springer Verlag, LNCS 714.

[JLMR94] Jourdan (M.), Lagnier (F.), Maraninchi (F.) et Raymond (P.). – A multiparadigm language for reactive systems. *In : In 5th IEEE International Conference on Computer Languages.* – Toulouse, mai 1994.

[JM94a] Jourdan (M.) et Maraninchi (F.). – A modular state/transition approach for programming real-time systems. *In : ACM Sigplan Workshop on Language, compiler and tool support for real-time systems.* – Orlando, FL, juin 1994.

[JM94b] Jourdan (M.) et Maraninchi (F.). – Studying synchronous communication mechanisms by abstractions. *In : IFIP Working Conference on Programming Concepts, Methods and Calculi.* – San Miniato, Italy, juin 1994.

[JM95] Jourdan (M.) et Maraninchi (F.). – Static timing analysis of real-time systems. *In : ACM Sigplan Workshop on Language, compiler and tool support for real-time systems.* – La Jolla, California, novembre 1995.

[JMO93] Jourdan (M.), Maraninchi (F.) et Olivero (A.). – Verifying quantitative real-time properties of synchronous programs. *In : International Conference on Computer-Aided Verification.* – Elounda, juin 1993.

[Jou91] Jourdan (M.). – *Etude des techniques de compilation applicables à un langage de description des systèmes réactifs.* – Dea, Grenoble, Université Joseph Fourier, Grenoble, juin 1991.

[Jou94] Jourdan (M.). – *Un environnement multi-langages et multi-outils pour la programmation et la vérification des systèmes réactifs.* – Thesis, Grenoble, Université Joseph Fourier, Grenoble, septembre 1994.

[Jou96] Jourdan (M.). – Integrating formal verification methods of quantitative real-time properties into a development environment for robot controllers. *In : AMAST Workshop on Real-time Systems.*

[LT89] Lynch (N. A.) et Tuttle (M. R.). – *An Introduction to Input/Output Automata.* – CWI-Quaterly n° 3, 1989.

[Mal93] Malik (S.). – Analysis of cyclic combinational circuits. *In : ICCAD'93.* – Santa Clara (Ca), 1993.

[Mar90] Maraninchi (F.). – *Argos : un langage graphique pour la conception, la description et la validation des systèmes réactifs.* – Thesis, Grenoble, Université Joseph Fourier, Grenoble, janvier 1990.

[Mar91] Maraninchi (F.). – The argos language: Graphical representation of automata and description of reactive systems. *In : IEEE Workshop on Visual Languages.* – Kobe, Japan, octobre 1991.

[Mar92] Maraninchi (F.). – Operational and compositional semantics of synchronous automaton compositions. *In : CONCUR.* – LNCS 630, Springer Verlag.

[MH96a] Maraninchi (F.) et Halbwachs (N.). – Compiling argos into boolean equations. *In : Formal Techniques for Real-Time and Fault Tolerance (FTRTFT).* – Uppsala (Sweden), septembre 1996.

[MH96b] Maraninchi (F.) et Halbwachs (N.). – Compositional semantics of non-deterministic synchronous languages. *In : European Symposium On Programming.* – Linkoping (Sweden), avril 1996.

[Mil80] Milner (R.). – A calculus of communication systems. *In : LNCS 92.* – Springer Verlag, 1980.

[Mil89] Milner (R.). – Communication and concurrency. *In : International Series in Computer Science.* – Prentice Hall, 1989.

[MV92a] Maraninchi (F.) et Vachon (M.). – *Argos Programming Examples: The Multi-Function, Six Digit LCD Watch Circuit With Alarm, Three Stopwatch Modes and Event Counter.* – Tr, LGI/IMAG, mars 1992.

[MV92b] Maraninchi (F.) et Vachon (M.). – An experience in compiling a mixed imperative/declarative language for reactive systems. *In : International Workshop on Compiler Construction (poster session).* – Springer Verlag, LNCS 641.

[PPST93] Pandey (Rajeev K.), Pesch (Wolfgang), Shur (Jim) et Takikawa (Masami). – *A Revised Leda Language Definition.* – Rapport technique n° 93-60-02, Department of Computer Science, Oregon State University, 1993. Fri, 15 Dec 1995 03:16:57 GMT.

[PS88] Pnueli (A.) et Shalev (M.). – *What is in a Step.* – Rapport technique, Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Israel, mai 1988.

[QV94] Quint (V.) et Vatton (I.). – Making structured documents active. *Electronic Publishing – Origination, Dissemination and Design*, vol. 7, n° 2, juin 1994, pp. 55–74.

[Rau92] Rauzy (A.). – *Toupie: Technical Report.* – Rapport technique, LaBRI-CNRS-Université Bordeaux I, juillet 1992.

[Ray96] Raymond (P.). – Recognizing regular expressions by means of dataflows networks. *In : 23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. – Springer Verlag.

[RdS90] Roy (V.) et de Simone (R.). – Auto and Autograph. *In : International Workshop on Computer Aided Verification*, éd. par Kurshan (R.). – Rutgers (N.J.), juin 1990.

[RM95] Rutten (E.) et Martinez (F.). – SignalGTI, implementing task preemption and time interval in the synchronous data-flow language SIGNAL. *In : 7th Euromicro Workshop on Real Time Systems.* – Odense (Denmark), juin 1995.

[Roc92] Rocheteau (F.). – *Extension du langage Lustre et application à la conception de circuits: Le langage Lustre-V4 et le système Pollux.* – Thèse, Institut National Polytechnique de Grenoble, juin 1992.

[Roy90] Roy (V.). – *Autograph, Un outil de visualisation pour les calculs de processus.* – Thèse, France, Université de Nice, 1990.

[RS89] Roy (V.) et Simone (R. de). – *An* Autograph *Primer.* – Rapport de recherche, Sophia-Antipolis, INRIA, mai 1989.

[Sch94] Schaar (P.). – *Un environnement de programmation pour le langage graphique Argos.* – Rapport technique, Conservatoire National des Arts et Métiers, mars 1994.

[SECK93] Simon (D.), Espiau (B.), Castillo (E.) et Kapellos (K.). – Computer-aided design of a generic robot controller handling reactivity and... *IEEE Transactions on Control Systems Technology*, vol. 1, 1993.

[SFLM93] Scholl (P.-C.), Fauvet (M.-C.), Lagnier (F.) et Maraninchi (F.). – *Cours d'informatique : langages et programmation.* – Masson, 1993.

[Shu91] Shur (Jim). – *Implementing Leda: Objects and Classes.* – Rapport technique n° 91-60-11, Department of Computer Science, Oregon State University, 1991. Fri, 15 Dec 1995 03:15:02 GMT.

[SKEJ95] Simon (D.), Kapellos (K.), Espiau (B.) et Jourdan (M.). – Formal verification of robotic missions and tasks. *In : European Workshop on Hybrid Systems.* – Grenoble, juin 1995.

[Smo94a] Smolka (Gert). – *A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards.* – Research Report n° RR-94-03, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, Deutsches Forschungszentrum für Künstliche Intelligenz, février 1994.

[Smo94b] Smolka (Gert). – A foundation for higher-order concurrent constraint programming. *In : 1st International Conference on Constraints in Computational Logics*, éd. par Jouannaud (Jean-Pierre). pp. 50–72. – München, Germany, 7–9 septembre 1994.

[Sto88] Stoughton (Allen). – *Fully Abstract Models of Programming Languages.* – Pitman, 1988, *Research Notes in Theoretical Computer Science.*

[SY96] Sifakis (J.) et Yovine (S.). – Compositional specification of timed systems. *In : 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96.* Lecture Notes in Computer Science 1046, pp. 347–359. – Grenoble, France, février 1996.

[US94] Uselton (A.C.) et Smolka (S.A.). – A process algebraic semantics for statecharts via state refinement. *In : IFIP Working Conference on Programming Concepts, Methods and Calculi.* – San Miniato, Italy, jun 1994.

[vdB94] von der Beeck (M.). – A comparison of statecharts variants. *In : FTRTFT.* – LNCS 863, Springer Verlag.

[VNG91] Vahid (F.), Narayan (S.) et Gajski (D.D.). – Speccharts: A language for system level synthesis. *In : Computer Hardware Description Languages.*

[ZJ93] Zave (Pamela) et Jackson (Michael). – Conjunction as composition. *acm Transactions of Software Engineering and Methodology*, vol. 2, n° 4, octobre 1993, pp. 379–411.

Sixième partie

Dossier

Chapitre 17

Curriculum Vitae

17.1 Informations personnelles

Nom :	Maraninchi Florence
Date et lieu de naissance :	9 octobre 1964 à Vif (Isère)
Nationalité :	Française
Situation de famille :	célibataire
Adresse personnelle :	8, rue Pierre Loti, 38000 Grenoble
Téléphone :	Bureau : 04 76 63 48 53

Fonctions :

- De novembre 1987 à octobre 1989, Allocataire de Recherche (MRT).
- D'octobre 1989 à octobre 1990, Allocataire d'Enseignement et de Recherche à l'Institut National Polytechnique de Grenoble (Ensimag).
- Depuis octobre 1990, Maître de Conférences à l'Université Joseph Fourier, Grenoble I (titularisation le 01/10/1991).

Etudes et titres universitaires :

- Juin 1982 : Baccalauréat Série C à Grenoble, mention Très Bien
- Juin 1984 : DEUG A (Sciences des Structures et de la Matière), Université Joseph Fourier (Grenoble I), mention Bien
- Juin 1985 : Licence d'Informatique, Université Joseph Fourier, mention Très Bien
- Juin 1986 : Maîtrise d'Informatique, Université Joseph Fourier, mention Bien
- Juin 1987 : DEA d'Informatique, "*Statecharts : sémantique et application à la spécification de systèmes*", Institut National Polytechnique, Grenoble. Responsable : M. Jacques Voiron. Mention Très Bien.
- Janvier 1990 : Thèse d'Informatique, "*Argos : un langage graphique pour la conception, la description et la validation des systèmes réactifs*", soutenue à Grenoble le 12 Janvier 1990, Université Joseph Fourier (Grenoble I). Mention Très Honorable.

Membres du jury :
M. Sacha Krakowiak, Professeur, Université J. Fourier, Grenoble (président).
M. Jacques Voiron, Professeur, Université J. Fourier, Grenoble (directeur)
M. Gérard Berry, Maître de Recherches, Ecole des Mines de Paris (rapporteur)
M. Michel Sintzoff, Professeur, Université Catholique de Louvain (rapporteur)
M. Joseph Sifakis, Directeur de Recherche CNRS, Grenoble.

17.2 Activités d'enseignement

17.2.1 Cadre

Après une année d'enseignement à l'ENSIMAG (algorithmique en première année et projet de compilation en deuxième année), mes activités d'enseignement se sont déroulées au sein de l'UFR IMA (Informatique et Mathématiques Appliquées) de l'Université Joseph Fourier, dans plusieurs contextes :

- dans la section MPI (Mathématiques, Physique, Informatique) de la deuxième année de DEUG A de 1990 à 1996, puis dans la filière MIAS (Mathématiques, Informatique et Applications aux Sciences) du DEUG rénové.

- en première année de l'Ecole Universitaire d'Informatique, modules L&P (Langages et Programmation) et ALM (Architectures Logicielles et Matérielles)

- en DESS Informatique Double Compétence, module Base de Données.

J'ai d'autre part participé à une action de formation permanente pour le personnel de la société Schneider Electric (anciennement Merlin-Gérin), mise en place par l'Université Joseph Fourier et l'Université Pierre Mendès-France, pour les années 93/94 et 94/95.

Les étudiants de la section MPI du DEUG suivaient 6h d'enseignement d'Informatique par semaine, pendant toute l'année, réparties en : 2h de cours, 2h de TD sur papier, 2h de TD sur machine, dits TD d'"expérimentation". Le contenu de [SFLM93] constituait le support des enseignements. Il en est de même actuellement pour les modules proposés dans le cadre de la réforme des premiers cycles, pour le DEUG mention MIAS.

Les enseignements en première année de l'Ecole sont divisés en trois blocs principaux : *Langages et Programmation*, *Architectures Logicielles et Matérielles* et *Outils Formels pour l'Informatique*. Les deux premiers blocs sont organisés en Cours, Travaux Dirigés et Travaux Dirigés dits "d'application". Les années 90/91 et 91/92 ont été des années de transition dans l'UFR, puisque l'ensemble des filières informatiques de second cycle (maîtrise classique, MST Experts en Systèmes Informatiques et MIAGE) ont été réorganisées pour former l'Ecole Universitaire d'Informatique. J'ai participé aux réflexions préliminaires dès le printemps 1990, et j'ai démarré mes enseignements au sein de l'Ecole dès la rentrée 90. Dans le cadre des Travaux Dirigés d'application du bloc Langages et Programmation, l'équipe d'enseignants a dû, lors de cette année d'installation de l'Ecole, définir un ensemble de "scénarios" d'expérimentation, sorte de TP courant sur 4 ou 5 semaines. J'ai ensuite participé aux enseignements du module ALM, en assurant une partie du cours (système à l'exécution, assemblage et édition de liens, systèmes de gestion de fichiers et de processus), en assurant les travaux dirigés de programmation en assembleur Sparc, et en définissant le projet de programmation d'un assembleur/éditeur de liens pour Sparc. Plus récemment j'ai participé aux travaux dirigés de description de circuits.

Le DESS IDC (Informatique Double Compétence) accueille des étudiants titulaires d'une maîtrise scientifique ou de diplômes plus élevés (DEA, plus rarement thèses), de domaines aussi différents que la biologie ou les mathématiques pures. L'hétérogénéité du public, ainsi que leur exigence d'une formation informatique suffisamment large, pose des problèmes de définition du contenu et de choix du niveau de discours. La forte motivation des étudiants et la diversité de leurs cultures scientifiques font de cette formation un cadre de travail très intéressant.

17.2.2 Rédaction de documents

Le travail en DEUG a donné lieu à un polycopié, puis à l'ouvrage [SFLM93]. Avec mes collègues du module ALM de première année de l'Ecole d'Informatique, nous avons commencé la rédaction d'un ouvrage intitulé 'Architectures Logicielles et Matérielles', qui devrait être prêt fin 98.

La définition de travaux pratiques et de projets de programmation donne lieu, par ailleurs, à la rédaction de documents.

17.2.3 Définition de travaux pratiques et projets de programmation

Dans le cadre du DEUG les travaux pratiques sont souvent organisés autour de programmes dont les étudiants complètent certains modules. Cela demande une préparation conséquente, et une écriture extrêmement rigoureuse des portions de programme effectivement lisibles par les étudiants.

L'essentiel de ma contribution (réalisée une première fois en Pascal sous environnement unix, et cette année en C/Unix) concerne :

- Une bibliothèque de fonctions (y compris l'affichage de cartes sur un écran graphique) pour programmer des réussites
- Un programme d'analyse lexicale et syntaxique d'un langage d'expressions qui décrit des motifs graphiques, complété par la construction du motif et l'affichage graphique
- Un programme de simulation d'automate

Dans le cadre de la première année de l'Ecole d'Informatique, l'essentiel de mon travail est la définition d'un projet de programmation en C d'un assembleur/éditeur de liens pour Sparc. Le programmme est entièrement réalisé et une version à compléter est utilisée par environ 120 étudiants chaque année depuis 4 ans. Le programme est écrit en utilisant l'outil noweb de programmation instruite ('literate programming'). La documentation complète comprend des chapitres de cours sur l'assemblage/édition de liens, et le programme à compléter, abondamment commenté. La version encore utilisée cette année produit des fichiers objet au format sun os. La disparition probable, dans un avenir proche, de la dernière machine fonctionnant encore sous ce système m'a incitée cette année à effectuer le portage à Solaris, ce qui a demandé une refonte assez conséquente du module de génération des fichiers objets.

17.3 Charges collectives

• Membre élu de la commission de spécialité des 5ème, 6ème et 27ème sections de l'UJF, depuis 1992.

• Responsable de la conception et du développement du serveur W3 de l'UFR Informatique et Mathématiques Appliquées (IMA) de l'UJF.

17.4 Activités de recherche

Les activités de recherche étant décrites en détail dans le présent document, nous ne parlons ici que du cadre de travail.

17.4.1 l'unité mixte VERIMAG

Mes activités de recherche se déroulent au sein de l'Unité Mixte de Recherche du CNRS n° C9939 VERIMAG qui regroupe le projet SPECTRE (Spécification et Programmation des Systèmes Communicants et Temps Réel) animé par Joseph Sifakis et la société VERILOG. Spectre est un projet IMAG et INRIA Rhône-Alpes.

L'objectif du projet est d'aider le concepteur de certains types d'applications parallèles et temps-réel pour lesquelles le développement d'une méthode de conception rigoureuse, appuyée par des outils automatiques, semble à la fois nécessaire et réaliste à l'heure actuelle. Les axes de recherche principaux sont : la programmation, la spécification et la validation de ces systèmes dits *critiques*.

17.4.2 Collaborations et relations nationales et internationales

- Les travaux de recherche autour des langages synchrones et de la vérification des systèmes ont conduit à des collaborations avec des équipes du même domaine (équipe d'André Arnold à Bordeaux, de Gérard Berry à Sophia-Antipolis, d'Albert Benveniste à Rennes) ou de domaines dans lesquels notre approche est utilisable (projet INRIA BIP dirigé par Bernard Espiau à Grenoble, équipe de Brigitte Plateau, avant-projet INRIA SHOOD dirigé par Toan Nguyen).

- Le projet SYNCHRONIE de la GMD (Allemagne) animé par A. Poigné, est consacré au développement d'un environnement de programmation synchrone multi-langages. Leur choix s'est porté sur Esterel, Lustre et Argos. La collaboration porte sur les aspects multi-langages, déjà étudiés dans la thèse de Muriel Jourdan.

- J'assiste très régulièrement depuis 1990 aux réunions du groupe C2A (Collaboration CAO Automatique) animé par Albert Benveniste, qui réunit des laboratoires publics français dans lesquels sont développés des langages synchrones (VERIMAG à Grenoble, IRISA à Rennes, INRIA à Sophia-Antipolis...) et des industriels intéressés par les aspects synchrones. Une des activités de ce groupe de travail a consisté à définir les formats de la "plateforme synchrone", en vue de la normalisation. Ce groupe a également donné naissance au projet européen Eureka SYNCHRON auquel je participe. Les partenaires sont : l'INRIA et les sociétés Vérilog, TNI et Schneider Electric (Merlin-Gérin) en France ; Saab Military Aircrafts et Logikkonsult en Suède.

- Je participe depuis octobre 1989 aux "Basic Research Actions" ESPRIT dans lesquelles le projet Spectre est engagé, d'abord SPEC, puis REACT. Parmi les partenaires on trouve les Universités de Liège, Kiel, Eindhoven, Oxford, Tampere (Finlande) et Uppsala (Suède) et le Weizmann Institute à Rehovot, Israël. Cela a donné lieu à de nombreux exposés (avril

et septembre 1990, septembre 1991, juin 1993, février et juillet 1994), dont 3 à l'occasion des revues de projets.

- Dans le cadre de la coopération Franco-Israélienne, j'ai effectué en mai 1991 un séjour d'une dizaine de jours au Weizmann Institute, Rehovot, Israël, sur l'invitation d'Amir Pnueli. La coopération portait sur des aspects de définition de la sémantique des langages synchrones impératifs. Une comparaison d'Argos avec diverses sémantiques des Statecharts a été développée à cette occasion.

- En novembre 1994 et décembre 1996 à Dagstuhl (Allemagne) et en novembre 1995 à Marseille, j'ai participé aux séminaires consacrés aux langages synchrones, sur l'invitation des organisateurs. Y participaient une quarantaine de personnes, représentant la majorité des équipes travaillant sur le sujet.

- En octobre 1995, j'ai participé à une réunion du groupe de travail WG2.1 de l'IFIP, à Ulm, en tant qu'observateur invité.

17.4.3 Encadrement d'étudiants

Année Universitaire 1990-1991 : Encadrement du **DEA** de Muriel Jourdan sur le sujet : *étude des techniques de compilation applicables à un langage de description des systèmes réactifs, le cas d'Argos.* La première partie du travail concerne la traduction d'Argos vers une des formes intermédiaires de la chaîne de compilation Esterel. Muriel Jourdan effectuait en parallèle avec son DEA un **magistère** d'Informatique à l'Université Joseph Fourier, formation pour laquelle j'étais également tuteur.

D'octobre 1991 à octobre 1994 : Encadrement de la **thèse** de Muriel Jourdan sur le sujet : *un environnement de programmation multi-langages et multi-outils pour les systèmes réactifs.* La soutenance a eu lieu le 29 septembre 1994. Les rapporteurs étaient MM Michel Sintzoff, Professeur à l'Université Catholique de Louvain, et Gérard Berry, Maître de Recherches à l'Ecole des Mines de Paris. Muriel Jourdan a ensuite effectué un stage post-doctoral dans l'équipe de Bernard Espiau (projet BIP, INRIA Rhône-Alpes) où elle a étudié les possibilités d'utilisation de nos outils pour l'analyse des systèmes de contrôle/commande en robotique. Elle est maintenant chargée de recherche à l'INRIA, dans le projet Opéra de V. Quint.

Année 1993 : Encadrement du mémoire du cycle C du **CNAM** de Philippe Schaar, sur le sujet : *Un environnement de programmation pour le langage graphique Argos.* L'idée était de réaliser un prototype d'éditeur syntaxique pour ARGOS basé sur l'utilisation de l'éditeur de documents structurés GRIF, développé au sein du projet OPERA de l'INRIA à Grenoble. Le prototype permet l'édition de programmes ARGOS sous forme graphique ou textuelle et la connexion au compilateur.

Année 1994/1995 : Tutorat de **Magistère** de Pascal Laffont, sur le sujet : *une étude de cas programmée en Argos et Lustre.*

De mars 96 à juin 96 : Encadrement du mémoire de DEA de l'Université de Iasi (Roumanie) de Traian Popovici, sur le thème : *Intégration de langages synchrones impératifs et déclaratifs : étude d'un processeur DC.*

Année 1996/1997 : Tutorat de **Magistère** de Yann Remond, sur le sujet : *Programmation multi-langages - Etude d'un mécanisme d'édition de liens de programmes DC.*

17.4.4 Cours, séminaires et jurys de thèses

Cours

- Dans le cadre du Programme Affilié de l'institut IMAG — programme de formation des industriels, auquel sont également conviés les chercheurs du laboratoire —, j'ai donné un cours assorti de démonstration du logiciel Argonaute, sur la programmation des systèmes réactifs dans les langages synchrones et en Argos en particulier (1989 et 1990).

- En mai 1993, j'ai organisé à Grenoble un cours de deux jours sur les langages synchrones et les méthodes et outils de vérification développés dans le laboratoire, à l'intention d'ingénieurs de la société Parallax Software Technologies.

- En mai 1993 également, j'ai participé, avec Paul Caspi et Xavier Nicollin, à un cours d'une journée sur l'approche synchrone, le langage Argos et les méthodes de vérification de propriétés temporelles, au CEA à Saclay.

- En mars 1995 j'ai été invitée par Lionel Marcé à Brest pour présenter le langage Argos et l'environnement de programmation-vérification Argonaute aux étudiants du DESS Informatique et Automatique de l'Université de Bretagne Occidentale. Cette présentation de 6h était divisée en Cours et TD.

Séminaires divers

- Le groupe C2A fournit l'occasion de présenter des résultats de recherche devant un public d'industriels et de chercheurs, éventuellement de culture différente, comme les automaticiens.

- J'ai participé en décembre 1993 à une réunion du groupe de travail Afcet sur la "sécurité des systèmes informatiques".

- Les invitations d'autres équipes sont généralement l'occasion de séminaires. J'ai ainsi présenté deux articles de conférence à l'équipe d'André Arnold (Bordeaux) en juin 94. Fabienne Lagnier et moi avons présenté l'approche synchrone, les langages Argos et Lustre et les aspects de vérification lors d'un séminaire de l'unité mixte Bull Imag en février 94.

Jurys de thèses, revues d'articles

J'ai participé en décembre 92 au jury de la thèse préparée par Guido Gherardi à l'Université de Nice sous la direction de Gérard Berry, en tant que co-rapporteur. En octobre 1994, j'ai été examinateur dans le jury de thèse de Frédéric Mignard (Ecole des Mines de Paris), également encadré par Gérard Berry.

Je participe régulièrement au travail de revue d'articles pour des conférences ou des revues d'audience internationale, dont AMAST (Algebraic Methods and Software Technologies), PROCOMET (Programming Concepts, Methods and Calculi), CONCUR (), ESOP (European Symposium on Programming), ECC (European Control Conference), FTRTFT (Formal Techniques for Real-Time and Fault-Tolerant Systems), IEEE TSE (Transactions on Software Engineering), SCP (Science of Computer Programming), ...

17.4.5 Développement de logiciels

Le développement de logiciels autour d'Argos et de l'environnement associé Argonaute a commencé en 1988. Le premier environnement argonaute était composé de :

- l'environnement d'édition réalisé en Xwindows version 10 par 4 étudiants de DESS Génie Informatique de Grenoble que j'encadrais en 1987-88.
- le debugger symbolique associé, que je développai par la suite en complétant l'éditeur.
- la première version du compilateur, développée en C dans le cadre de ma thèse, après une tentative de prototypage en Lisp.

Muriel Jourdan, en thèse de 91 à 94, a réalisé quelques améliorations, puis la connexion à une bibliothèque de bdd, pour tous les aspects de calcul booléen dans les programmes. Elle a implanté la traduction en format IC (un format intermédiaire de la chaîne de compilation Esterel) et une technique d'élimination du raffinement. Elle a pris en charge la connexion à Kronos, et le petit prototype de traduction Argos vers Lustre .

J'ai travaillé pour ma part aux connexions du compilateur aux outils Bac (analyse de causalité sur la forme équationnelle) et Polka (analyse de systèmes hybrides) de Nicolas Halbwachs.

Plus récemment, j'ai démarré une refonte complète du compilateur et de ses connexions aux divers outils de vérification, en passant par le format DC. J'ai donc entrepris la programmation en C++ et noweb du compilateur à partir d'une syntaxe qui autorise la structuration des programmes en "processus", analogues aux noeuds Lustre.

L'ambition de ce projet est de réaliser un environnement de programmation autour d'un petit langage permettant de composer des automates, décorés d'informations diverses selon les besoins. Cet environnement utilisera DC comme format intermédiaire, pour permettre l'édition de liens avec des programmes Lustre ou Esterel.

17.5 Publications

[HM95] N. Halbwachs et F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. Dans *EUROMICRO*, Como, Italy, septembre 1995.

[HMP95] N. Halbwachs, F. Maraninchi et Y. E. Proy. The railroad crossing problem, modeling with hybrid argos - analysis with polka. Dans *Second European Workshop on Real-Time and Hybrid Systems*, Grenoble (France), juin 1995.

[JLMR93] M. Jourdan, F. Lagnier, F. Maraninchi et P. Raymond. Embedding declarative subprograms into imperative constructs. Dans *Fifth International Symposium on Programming Language Implementation and Logic Programming,* Tallin, Estonia. Springer Verlag, LNCS 714, août 1993.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi et P. Raymond. A multiparadigm language for reactive systems. Dans *In 5th IEEE International Conference on Computer Languages*, Toulouse, mai 1994. IEEE Computer Society Press.

[JM94a] M. Jourdan et F. Maraninchi. A modular state/transition approach for programming real-time systems. Dans *ACM Sigplan Workshop on Language,*

compiler and tool support for real-time systems, Orlando, FL, juin 1994. `http://www.cs.umd.edu/~pugh/sigplan_realtime_workshop_94/`.

[JM94b] M. Jourdan et F. Maraninchi. Studying synchronous communication mechanisms by abstractions. Dans *IFIP Working Conference on Programming Concepts, Methods and Calculi*, San Miniato, Italy, juin 1994. Elsevier Science Publishers.

[JM95] M. Jourdan et F. Maraninchi. Static timing analysis of real-time systems. Dans *ACM Sigplan Workshop on Language, compiler and tool support for real-time systems*, La Jolla, California, novembre 1995. ACM Sigplan Notices.

[JM96] M. Jourdan et F. Maraninchi. Vérification de systèmes réactifs en argos temporisé. Dans *Congrès AFCET : Modélisation des systèmes réactifs*, Brest (France), mars 1996.

[JMO93] M. Jourdan, F. Maraninchi et A. Olivero. Verifying quantitative real-time properties of synchronous programs. Dans *International Conference on Computer-Aided Verification*, Elounda, juin 1993. LNCS 697.

[Mar87] F. Maraninchi. Statecharts, sémantique et application à la spécification de systèmes. DEA, LGI/IMAG, Grenoble, Grenoble, juin 1987.

[Mar89] F. Maraninchi. Argonaute, graphical description, semantics and verification of reactive systems by using a process algebra. Dans *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, juin 1989. LNCS 407, Springer Verlag.

[Mar90] F. Maraninchi. Argos : un langage graphique pour la conception, la description et la validation des systèmes réactifs. Thesis, Université Joseph Fourier, Grenoble, Grenoble, janvier 1990.

[Mar91] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. Dans *IEEE Workshop on Visual Languages*, Kobe, Japan, octobre 1991.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. Dans *CONCUR*. LNCS 630, Springer Verlag, août 1992.

[MH96a] F. Maraninchi et N. Halbwachs. Compiling argos into boolean equations. Dans *Formal Techniques for Real-Time and Fault Tolerance (FTRTFT)*, Uppsala (Sweden), septembre 1996. Springer verlag, LNCS 1135.

[MH96b] F. Maraninchi et N. Halbwachs. Compositional semantics of non-deterministic synchronous languages. Dans *European Symposium On Programming*, Linkoping (Sweden), avril 1996. Springer verlag, LNCS 1058.

[MSFL93] F. Maraninchi, P.-C. Scholl, M.-C. Fauvet et F. Lagnier. Rôle de l'expression fonctionnelle dans l'enseignement de l'informatique en deug a. Dans *2èmes journées de travail : les langages applicatifs dans l'enseignement de l 'informatique*, Rennes, apr 1993. SPECIF.

[MV92a] F. Maraninchi et M. Vachon. Argos programming examples: The multi-function, six digit lcd watch circuit with alarm, three stopwatch modes and event counter. TR, LGI/IMAG, mars 1992.

[MV92b] F. Maraninchi et M. Vachon. An experience in compiling a mixed imperative/declarative language for reactive systems. Dans *International Workshop on Compiler Construction (poster session)*. Springer Verlag, LNCS 641, octobre 1992.

[SFLM93] P.-C. Scholl, M.-C. Fauvet, F. Lagnier et F. Maraninchi. *Cours d'informatique : langages et programmation*. Masson, 1993.

17.6 Documents pédagogiques

P.-C. Scholl, M.-C. Fauvet, F. Lagnier et F. Maraninchi, *Cours d'informatique : Langages et Programmation*, Ouvrage édité par Masson, 1993 (420 pages). Préface de Michel Sintzoff.

J.-P. Peyrin, C. Berrut et F. Maraninchi, *Langages et Programmation, rappels de cours et exercices*, polycopié pour la première année de l'Ecole Universitaire d'Informatique, 150 pages, tiré à 150 exemplaires.

C. Berrut et F. Maraninchi, *Introduction à Lisp*, polycopié, support de cours-TD pour la première année de l'EUI, 100 pages

C. Collet et F. Maraninchi, *Mini interprète Lisp*, Documentation enseignants du scénario de programmation d'un mini-interprète Lisp, support de TD pour la première année de l'EUI

P.-C. Scholl, M.-C. Fauvet, F. Lagnier et F. Maraninchi, *Informatique, exemples et exercices*, polycopié, support de cours-TD pour l'option MPI (Mathématiques, Physique et Informatique) du DEUG A 2ème année.

P.-C. Scholl, M.-C. Fauvet, F. Lagnier et F. Maraninchi, *Algorithmique, exemples et exercices*, polycopié, support de cours-TD pour le DESS Informatique Double Compétence.

F. Maraninchi, *Projet Assembleur/Editeur de liens pour Sparc, définition et documentation*, 50p. Ce document comprend une description complète du langage d'assemblage à traiter (lexicographie, syntaxe et sémantique) ainsi que les algorithmes principaux des phases d'analyse, génération de code et édition de liens. Une annexe décrit la structure du logiciel à compléter fourni aux étudiants.

Chapitre 18

Publications jointes

On trouvera ci-joint, et dans cet ordre :

[Mar92]	Première version de la sémantique, réécrite au chapitre 2
[JM94b]	Source du chapitre 3
[MH96]	Source du chapitre 4
[HM95]	Complétant la section 5.2 du chapitre 5
[JLMR94]	Complétant le chapitre 7
[JM94a]	Un exemple de programme ARGOS.
[JMO93]	Complétant le chapitre 10.

Rappel des références

[Mar92] Maraninchi (F.). – Operational and compositional semantics of synchronous automaton compositions. *In : CONCUR.* – LNCS 630, Springer Verlag.

[JM94b] Jourdan (M.) et Maraninchi (F.). – Studying synchronous communication mechanisms by abstractions. *In : IFIP Working Conference on Programming Concepts, Methods and Calculi.* – San Miniato, Italy, juin 1994.

[MH96] Maraninchi (F.) et Halbwachs (N.). – Compositional semantics of non-deterministic synchronous languages. *In : European Symposium On Programming.* – Linkoping (Sweden), avril 1996.

[HM95] Halbwachs (N.) et Maraninchi (F.). – On the symbolic analysis of combinational loops in circuits and synchronous programs. *In : EUROMICRO.* – Como, Italy, septembre 1995.

[JLMR94] Jourdan (M.), Lagnier (F.), Maraninchi (F.) et Raymond (P.). – A multiparadigm language for reactive systems. *In : In 5th IEEE International Conference on Computer Languages.* – Toulouse, mai 1994.

[JM94a] Jourdan (M.) et Maraninchi (F.). – A modular state/transition approach for programming real-time systems. *In : ACM Sigplan Workshop on Language, compiler and tool support for real-time systems.* – Orlando, FL, juin 1994.

[JMO93] Jourdan (M.), Maraninchi (F.) et Olivero (A.). – Verifying quantitative real-time properties of synchronous programs. *In : International Conference on Computer-Aided Verification.* – Elounda, juin 1993.

Document d'Habilitation à Diriger des Recherches
Université Joseph Fourier - Grenoble I
Modélisation et validation des systèmes réactifs :
un langage synchrone à base d'automates
Florence Maraninchi

Résumé

Ce document décrit la conception du langage Argos. C'est un langage synchrone à syntaxe graphique, qui permet la description de systèmes réactifs sous forme de compositions de machines de Mealy. Le jeu d'opérateurs est en quelque sorte minimal : mise en parallèle, synchronisation par le mécanisme de diffusion synchrone commun à tous les langages synchrones, composition hiérarchique héritée des Statecharts. La composition parallèle de machines à états et la composition hiérarchique rendent réaliste la description de systèmes de taille conséquente par des systèmes d'états et de transitions explicites.

Nous décrivons la démarche de conception du langage : problèmes liés au choix des constructions et à leur sémantique ; implantation efficace ; connexions du compilateur à divers outils de vérification du domaine ; génération de code. Nous présentons également les extensions du langage proposées pour la spécification des systèmes temporisés, hybrides ou non-déterministes, et les expériences de programmation multi-langages menées dans le cadre synchrone avec Esterel et Lustre.

Nous montrons les relations étroites entre la définition d'Argos et certains aspects de l'enseignement de l'algorithmique en Deug.
Mots-clé:
Systèmes réactifs, langages synchrones, modélisation, programmation, validation, automates, Argos, algorithmique

Abstract

This document is about the design of the Argos language. It's a synchronous language, in which reactive systems are described as compositions of Mealy machines. The set of operators is in some sense minimal: the parallel composition; the synchronous broadcast common to all synchronous languages, as the communication mechanism; a hierarchic composition inspired from Statecharts. The parallel and hierarchic compositions allow the description of large systems bt means of explicit states and transitions.

We described the design phases of the language: choice of the construct, and definition of their semantics; efficient implementation; connections of the compiler to a wide variety of available verification tools; code generation. We also mention the extensions towards timed, hybrid or non-deterministic systems, and the experience in multi-language programming in the synchronous framework: Argos+Lustre or Argos+Esterel.

We show relationships between the definition of Argos and the teaching of algorithmics.
Keywords:
Reactive Systems, synchronous languages, modelling, programming, validation, automata, Argos, algorithmics