Défense et illustration des algèbres de processus

Hubert Garavel

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
F-38330 Montbonnot Saint-Martin
Hubert.Garavel@inria.fr

RÉSUMÉ.
Les algèbres de processus sont un formalisme mathématique pour la description et l'étude des systèmes concurrents. Dans cet article, nous expliquons pourquoi les concepts fondamentaux des algèbres de processus, lorsqu'ils sont combinés avec des langages appropriés pour la description des types de données, fournissent des solutions techniquement supérieures à d'autres formalismes. En particulier, nous soulignons les avantages intrinsèques des algèbres de processus sur quatre points généralement antinomiques : l'expressivité offerte aux utilisateurs, la capacité à modéliser divers types d'applications, la possibilité de générer automatiquement des implémentations prototypes et l'efficacité permise aux outils de vérification automatisée.

MOTS-CLÉS : algèbre de processus, calcul de processus, CCS, CSP, E-LOTOS*, indéterminisme, modèle, modélisation, langage de spécification,* LOTOS*, parallélisme, processus, spécification formelle, système asynchrone, type de données, vérification, vérification d'équivalence, vérification de modèle.*

1. Introduction

Au-delà des programmes *séquentiels*, qui prennent des données en entrée et fournissent des résultats en sortie, beaucoup de systèmes informatiques sont *parallèles*, c'est-à-dire composés de plusieurs *processus* qui s'exécutent simultanément et s'échangent des informations. Le parallélisme est dit *synchrone* lorsque le système possède une horloge globale qui, à intervalles réguliers, pilote l'exécution des processus. Dans le cas contraire, le parallélisme est dit *asynchrone* : chaque processus est alors libre d'évoluer à son propre rythme, mais doit se synchroniser avec d'autres processus lorsqu'il veut accéder à des ressources partagées, afin d'assurer une cohérence globale.

L'étude du parallélisme asynchrone trouve de nombreuses et importantes applications dans les domaines du logiciel, du matériel et des télécommunications. Historiquement, les systèmes distribués ont été à l'origine des premiers travaux ; puis l'apparition des réseaux et des protocoles de télécommunications a fourni — et fournit encore — de nouveaux problèmes ; enfin, le parallélisme asynchrone a gagné le monde des circuits, tant au niveau des architectures multiprocesseurs (arbitrage de bus, cohérence de caches...) qu'à l'intérieur des processeurs eux-mêmes (système sur puce, réseau sur puce...), une évolution renforcée par l'intérêt pour les circuits asynchrones, plus rapides et économes en énergie que leurs prédécesseurs synchrones.

En règle générale, les systèmes asynchrones sont plus difficiles à concevoir et à mettre au point que les systèmes séquentiels ou synchrones. En effet, l'amélioration continue des méthodologies et des langages de programmation permet aujourd'hui de construire des systèmes séquentiels complexes à peu près corrects — le niveau de correction atteint étant essentiellement déterminé par un problème de coût et d'expertise — grâce notamment à des concepts informatiques tels que le typage, la programmation structurée, les modules, les objets, l'analyse statique, les preuves formelles... De même, pour les systèmes synchrones, l'apport de langages dédiés [HAL 93] et d'outils pour la génération de code et la vérification permet de répondre aux exigences de sécurité pour des applications critiques. En revanche, la situation des systèmes asynchrones est beaucoup moins avancée, ceci pour plusieurs raisons :

– Les systèmes asynchrones sont intrinsèquement *indéterministes*, c'est-à-dire qu'une exécution donnée peut ne pas être prédictible ni reproductible, d'où des difficultés pour la mise au point et le test. Un corollaire de l'indéterminisme est qu'un système asynchrone peut, à certains instants, avoir plusieurs évolutions futures, généralement avec une combinatoire exponentielle qui limite fortement les possibilités de vérification automatisée (problème dit de l'*explosion d'états*).

– Les systèmes asynchrones n'ont pas de propriétés évidentes de *compositionnalité*, c'est-à-dire qu'il n'est pas aisé de construire un système correct par assemblage de sous-systèmes plus simples et prouvés corrects. Ainsi, la mise en parallèle de deux processus qui ne comportent pas de blocage peut cependant créer un interblocage. De fait, il n'existe pratiquement aucune propriété utile qui soit préservée par la mise en parallèle, ce qui rend difficile l'instauration d'un mécanisme de «typage» qui garantirait, dans un cadre suffisamment général, la correction des systèmes parallèles par construction.

– Peut-être à cause de la présence d'indéterminisme et de l'absence de compositionnalité, il semble que le cerveau humain ait du mal à appréhender les systèmes asynchrones, même lorsqu'il s'agit de systèmes relativement simples avec peu de processus. Ainsi, il n'est pas rare que les premières versions de protocoles innovants comportent des erreurs ayant échappé à l'attention de leurs concepteurs (par exemple, [NEE 78] et [LOW 95], [LAN 77] et [GAR 97]...)

Pour surmonter ces difficultés, d'importantes recherches ont été effectuées, depuis au moins trois décennies et dans plusieurs directions. La première d'entre elles porte sur la modélisation correcte des systèmes asynchrones par assemblage de processus décrits séparément. Ceci nécessite des formalismes adaptés permettant d'exprimer directement le parallélisme asynchrone, un concept absent des langages de programmation usuels (à l'exception d'ADA et, dans une moindre mesure, de JAVA) et des méthodes formelles conçues pour les programmes séquentiels (types abstraits algébriques, VDM [ISO 95], Z [ISO 02], B [ABR 96, HAB 01]...).

Un grand nombre de formalismes ont été proposés pour les systèmes asynchrones, parmi lesquels on peut mentionner les modèles à mémoire partagée (sémaphores, moniteurs, régions critiques), les réseaux de Petri et leurs multiples extensions, les automates communicants et leurs langages dérivés (*statecharts* [HAR 87], ESTELLE [ISO 88], *IO automata* [LYN 96, chap. 8], SDL [ITU 99], UML [OMG 03]...) ainsi que les *algèbres de processus* (aussi appelées *calculs de processus*) [CLE 99, FOK 00, BER 01]... D'un point de vue rationnel, un formalisme d'avenir devrait satisfaire quatre critères essentiels :

– *Expressivité :* le formalisme doit permettre de modéliser simplement, sans contorsions inutiles, les caractéristiques essentielles des systèmes asynchrones.

– *Universalité :* le formalisme doit être utilisable dans tous les domaines d'activité où intervient le parallélisme asynchrone (logiciel, matériel, télécommunications) et ne pas être étroitement dépendant d'un domaine particulier (comme ESTELLE et SDL l'ont été pour les télécommunications), ceci pour permettre des économies d'échelle dans le développement d'outils.

– *Exécutabilité :* le formalisme doit avoir un caractère exécutable afin que les modélisations de systèmes asynchrones ne servent pas seulement de documentation, mais puissent être traitées par des outils de simulation (pour effectuer la mise au point), de prototypage rapide (pour produire du code exécutable) et de génération automatique de tests.

– *Vérifiabilité :* certains systèmes asynchrones (notamment dans le domaine du matériel) sont suffisamment critiques pour que leur correction doive être garantie par des techniques de vérification ou de preuve. Or celles-ci ne peuvent être appliquées que sur des modélisations faites dans des formalismes dont la sémantique est, d'une part, rigoureusement définie, mais possède aussi de bonnes propriétés de compositionalité et d'abstraction permettant de repousser les limites de l'explosion d'états.

Nous sommes convaincus — et c'est le propos du présent article — que les algèbres de processus constituent, avec leurs plus récentes évolutions, le formalisme le plus proche des critères ci-dessus. Issue des travaux fondateurs de C.A.R. Hoare et R. Milner — tous deux lauréats du prix Türing — l'école des algèbres de processus a produit des résultats théoriques profonds, à commencer par une vision unifiée, élégante et féconde du parallélisme asynchrone. Ces idées ont conduit à la définition de multiples langages, trop nombreux pour qu'on les cite tous ici : CSP [HOA 78], CCS [MIL 80, MIL 89], TCSP [BRO 84, HOA 85, ROS 97, SCH 99], ACP [BER 84, BER 85], FP2 [JOR 84], MEIJE [de 85], CIRCAL [MIL 85], LCS [BER 94, BER 95, BER 96], LOTOS [ISO 89], PSF [MAU 90, MAU 93], μCRL [GRO 95, GRO 97], FSP [MAG 99b], E-LOTOS [ISO 01], LOTOS NT [SIG 00]...

Deux de ces langages, LOTOS et E-LOTOS, ont le statut de normes internationales ISO. Plusieurs langages de description de circuits s'inspirent directement des algèbres de processus. Il existe enfin de nombreuses extensions, notamment les algèbres synchrones, mobiles, temporisées, stochastiques, probabilistes...

Il est difficile de résumer en quelques phrases les caractéristiques essentielles des algèbres de processus. Dans son acception initiale, une algèbre de processus est un «petit» langage défini par une *syntaxe* et une *sémantique* :

– La syntaxe est à l'origine très simple, dans le style du λ-calcul, et comprend un nombre réduit d'opérateurs algébriques primitifs (composition séquentielle, choix indéterministe, composition parallèle..) qui, par assemblage, permettent de décrire des comportements complexes. Ainsi, un système asynchrone est-il décrit par un terme algébrique. Toutefois, les algèbres de processus ont progressivement évolué pour devenir des langages à part entière. De ce fait, leur syntaxe s'est enrichie de diverses façons, par exemple pour permettre la manipulation de données typées, pour favoriser la modularité des spécifications... En particulier, la récente norme E-LOTOS change radicalement l'apparence syntaxique en se rapprochant des langages de programmation fonctionnels et impératifs, ce qui ne remet pas en cause les principes sous-jacents.

– La sémantique est définie formellement, de manière *axiomatique* ou *opérationnelle*. Une sémantique axiomatique consiste en un ensemble de lois algébriques (commutativité, associativité, distributivité des opérateurs...) qui permettent de démontrer l'équivalence de termes. Une sémantique opérationnelle consiste en une *relation de transition* $B \xrightarrow{L} B'$ exprimant le fait qu'un terme B peut effectuer l'action L puis évoluer et se transformer en un terme B' ; cette relation de transition est généralement définie par induction structurelle sur la syntaxe des termes en utilisant des formats de règles standards [ACE 01] qui garantissent par construction que la sémantique est correcte ; elle détermine implicitement une correspondance entre un terme B et un automate qui décrit les évolutions futures de B (les transitions de cet automate étant étiquetées par les actions L effectuées par B) ; il est ainsi possible d'exécuter les termes algébriques et de vérifier leur correction en analysant l'automate qui leur correspond.

Cet article est structuré en deux grandes parties. La section 2 présente et évalue les principales approches pour la modélisation des données dans les systèmes asynchrones. La section 3 expose les concepts fondamentaux des algèbres de processus en les comparant aux autres formalismes existants.

2. Spécification des données

Initialement, de nombreux formalismes conçus pour le parallélisme asynchrone ne permettaient pas de modéliser les données présentes dans les systèmes. On peut ainsi mentionner les réseaux de Petri, les automates d'Arnold-Nivat [ARN 92]... Quant aux algèbres de processus, bien que CSP ait été dès l'origine capable de manipuler des données, de nombreux chercheurs ont préféré travailler sur des algèbres de processus sans données (*pure* CCS, *basic* LOTOS...)

Aujourd'hui, les principaux formalismes permettent de décrire les données aussi bien que les processus. De multiples approches ont été proposées pour la modélisation des données :

– importation de types et de fonctions externes, c'est-à-dire définis séparément (par exemple, en langage C dans le cas d'ESTEREL [BER 92]) ;

– utilisation de jetons «colorés» dans les réseaux de Petri ;

– utilisation de langages impératifs (sous-ensemble de PASCAL pour ESTELLE, variante de C pour PROMELA [HOL 91, HOL 97]) ;

– utilisation de types abstraits algébriques (FP2, LOTOS, SDL, μCRL) ;

– utilisation de langages fonctionnels (LCS, E-LOTOS).

Avec le recul du temps, ces diverses approches peuvent être évaluées en se référant aux quatre critères énoncés plus haut :

– Les critères d'expressivité et d'universalité conduisent à réduire l'écart entre la modélisation formelle des données d'un système asynchrone et leur implémentation réelle. En ce sens, les réseaux de Petri colorés semblent trop restrictifs et on doit plutôt reprendre des concepts usuels de types, variables, expressions, fonctions...

– Le critère d'exécutabilité implique également que la sémantique choisie pour modéliser les données ne soit pas trop éloignée de celle des langages de programmation usuels, d'une part, pour permettre une correspondance directe entre spécification formelle et implémentation et, d'autre part, pour permettre une génération de code simple et efficace. C'est ainsi que la sémantique équationnelle des types abstraits algébriques a été progressivement abandonnée et remplacée, soit par une sémantique basée sur la réécriture avec constructeurs et filtrage (comme pour certains outils pour LOTOS [GAR 89b] et pour la version révisée de μCRL [GRO 97]), soit par une sémantique inspirée des langages fonctionnels (comme pour LCS et E-LOTOS).

– Le critère de vérifiabilité a plusieurs conséquences. D'une part, il conduit à rejeter les importations de types externes lorsque le langage externe utilisé pour décrire les types n'a pas de sémantique formelle ; par ailleurs, si les importations de types sont parfois utiles (par exemple, pour des raisons de performances), leur utilisation systématique est source de problèmes pratiques, notamment pour la mise au point qui doit être faite à l'interface de deux langages distincts. D'autre part, ce critère favorise naturellement le choix d'un *typage fort*, qui permet la détection d'erreurs statiquement et que la plupart des formalismes ont adopté (sauf cas isolés comme ERLANG [ARM 93] et Tla+ [LAM 94, LAM 02]) : on peut donc raisonnablement exiger que les variables soient typées, ainsi que les arguments et le résultat des fonctions, et qu'il n'existe pas de conversion de types capable de subvertir les règles de typage.

Enfin, il impose que la sémantique des expressions et des fonctions soit formellement définie, ce qui n'est pas le cas de langages impératifs tels que C, C++, voire JAVA ; on se situe ici au point de démarcation entre les méthodes formelles et d'autres approches qui opèrent directement sur des langages de programmation (*software model checking*).

Dans cette section, nous tentons d'esquisser les caractéristiques essentielles d'un langage «idéal» non seulement pour la modélisation des données, mais aussi pour les techniques de preuve et de vérification automatisée (*model checking*) qui imposent leurs propres contraintes. La réflexion s'appuie sur les études effectuées lors de la normalisation du langage E-LOTOS (en particulier [JEF 95]), mais la prolonge sur certains points et s'en écarte sur d'autres. Nous commençons par une remarque préliminaire.

Lorsqu'un formalisme permet de décrire les données et les processus avec des mécanismes bien distincts (comme cela est souvent le cas, par exemple en ESTELLE, LOTOS, SDL...) il se pose quelquefois un problème de frontière : il n'est pas toujours simple de décider si telle partie d'un système asynchrone doit être représentée comme une donnée ou comme un processus, et l'avis d'un expert est alors utile pour déterminer la meilleure modélisation. Ce problème disparaît si l'on rapproche, syntaxiquement et sémantiquement, les mécanismes de description pour les données et pour les processus (comme c'est le cas en E-LOTOS où une fonction peut être vue comme un cas particulier de processus) : l'utilisateur peut alors aisément, par quelques changements syntaxiques, remettre en cause ses choix de modélisation initiaux. Dans ce contexte, la différence entre fonctions et processus apparaît essentiellement comme un facteur de lisibilité et une aide apportée aux outils (pour améliorer la génération de code, simplifier les preuves...).

2.1. *Choix liés à l'évaluation des expressions*

Nous examinons d'abord cinq problèmes essentiels concernant la définition sémantique des expressions et nous étudions leur impact sur la modélisation, l'implémentation et la vérification des systèmes asynchrones.

Non-terminaison

La présence de fonctions récursives et/ou de constructions itératives «**while**» fait qu'en général l'évaluation d'une expression peut ne pas terminer. Cette difficulté était habilement évitée avec les sémantiques équationnelles pour les types abstraits, puisque ces sémantiques ne cherchent pas à évaluer les expressions, mais seulement à déterminer si deux expressions peuvent être démontrées égales en un nombre fini de pas. Mais, après l'abandon progressif des approches équationnelles, le problème de la non-terminaison se pose de la manière suivante : soit on choisit de n'accepter que les programmes dont on peut prouver qu'ils terminent, ce qui conduit à imposer des restrictions (conditions suffisantes de terminaison) sur la classe des programmes autorisés (par exemple, en encadrant l'usage de la récursivité selon la théorie des fonctions primitives récursives, et en exigeant que chaque itération soit accompagnée d'une fonction de mesure décroissante), avec le risque de rejeter des programmes corrects et utiles ; soit on choisit d'accepter tous les programmes, avec l'inconvénient d'avoir une classe de programmes auxquels on ne sait pas donner de sémantique opérationnelle (bien que l'on puisse leur donner une sémantique dénotationnelle). En E-LOTOS, c'est la seconde solution qui a été choisie : la sémantique utilise un prédicat «*l'expression E peut s'évaluer à la valeur V en un nombre fini de pas*» qui, dans le cas général, est semi-décidable (on peut démontrer qu'il est vrai en un nombre fini de pas, mais démontrer qu'il est faux peut demander un nombre infini de pas).

Fonctions partiellement définies

On doit souvent modéliser des expressions dont la valeur n'est pas définie : débordement arithmétique, division par zéro, extraction de la tête d'une liste vide... Or, les formalismes existants n'autorisent, pour la plupart, que les fonctions *totales*, c'est-à-dire définies pour toutes les valeurs de leurs arguments. Ceci oblige l'utilisateur, soit à rajouter des résultats supplémentaires aux fonctions (ce qui diminue la lisibilité puisque chaque appel de fonction doit être suivi d'un test), soit à étendre les domaines des types par des valeurs spéciales servant à coder les cas d'erreur des fonctions (ce qui augmente le coût des données en mémoire et pénalise la vérification énumérative). La meilleure approche consiste à autoriser les fonctions *partielles* en adoptant le mécanisme d'*exceptions* qui existe dans les langages de programmation modernes (ADA, ML, JAVA) repris ensuite dans certaines algèbres de processus comme LCS et E-LOTOS.

Déterminisme des expressions

Si l'on peut concevoir des formalismes où l'évaluation des expressions serait indéterministe (c'est-à-dire qu'une même expression, évaluée dans un même environnement, puisse renvoyer des résultats différents et/ou lever des exceptions différentes), la traduction manuelle ou automatique de tels formalismes vers des langages de programmation usuels serait certainement complexe et peu performante. Il semble donc préférable d'adopter les choix des langages de programmation — eux-mêmes basés sur les jeux d'instructions des processeurs, qui sont prévus et optimisés pour des expressions déterministes — et supposer que, dans un environnement donné, une expression renvoie toujours le même résultat ou bien lève la même exception. Exclu de la modélisation des données, l'indéterminisme peut néanmoins apparaître dans les processus. Ainsi, la plupart des algèbres comportent un opérateur de choix pour exprimer le fait qu'une variable X prend une valeur quelconque dans un domaine T (ce qui s'écrit «**choice** $X : T$» en LOTOS, «$X :=$ **any** T» en E-LOTOS...)

Absence d'effets de bord

Pour des raisons d'efficacité, beaucoup de langages de programmation comportent des variables *globales* (ou *statiques*) qui, lorsque l'on évalue une expression ou qu'on appelle une fonction, peuvent être consultées voire modifiées. Dans les formalismes de modélisation, les variables globales ont plusieurs inconvénients. D'une part, elles portent atteinte au principe de déterminisme, puisque deux appels successifs d'une même fonction avec les mêmes paramètres peuvent renvoyer des résultats distincts. D'autre part, elles posent des problèmes sémantiques lorsqu'elles apparaissent dans les gardes booléennes qui autorisent les communications (une situation que le langage ESTELLE prohibait en imposant à certaines fonctions d'être *pures*, c'est-à-dire sans effet de bord). Compte tenu que les variables globales ont, en fait, une existence «parallèle» à l'évaluation des expressions, il est raisonnable de les réserver à la partie processus, où elles pourront être fidèlement modélisées. Pour les mêmes raisons, on peut interdire aux expressions d'effectuer des opérations d'entrée/sortie, ces dernières pouvant être aisément décrites dans la partie processus avec les moyens standards de communication.

Initialisation des variables

Si l'on veut un formalisme ayant une sémantique précise avec une évaluation déterministe des expressions, alors on doit garantir que toute variable est bien affectée avant d'être utilisée. Autrement, il existerait toute une classe de programmes auxquels on ne saurait pas donner de signification définie, et pire, on serait incapable d'identifier statiquement cette classe puisque le problème est indécidable. Si l'on s'autorisait des expressions indéterministes, on pourrait poser qu'une variable non initialisée représente l'ensemble des valeurs de son domaine, mais il est préférable, lorsque l'on veut introduire de l'indéterminisme, de le faire explicitement à l'aide des opérateurs de choix mentionnés ci-dessus, afin de bien distinguer l'indéterminisme intentionnel des inadvertances de modélisation. Dans les langages fonctionnels, toutes les variables sont initialisées, puisque la syntaxe ne permet pas de dissocier déclaration de variable et affectation. Pour les langages impératifs, une solution simple est de forcer l'initialisation des variables au moment de leur déclaration (comme en EIFFEL), mais il existe des méthodes plus fines (comme en HERMÈS [STR 86, STR 90], puis JAVA et LOTOS NT), basées sur l'analyse statique du flot des données, qui donnent des conditions suffisantes pour garantir que chaque variable, sur chaque chemin d'exécution, est affectée avant d'être lue ; le problème étant indécidable, on court le risque de rejeter des programmes corrects qui ne satisferaient pas ces conditions suffisantes ; néanmoins, comme le typage, ces contraintes statiques semblent bien acceptées par les utilisateurs ; elles imposent une discipline de programmation qui accroît la lisibilité et la qualité ; enfin, l'analyse de flot peut être affinée pour accepter davantage de programmes corrects.

2.2. *Choix liés à la représentation des types*

Lorsqu'il s'agit de modéliser des systèmes réels, tous les types de données présents dans les langages de programmation ont leur utilité. Cependant, toutes les approches employées pour la définition des types ne sont pas «égales» : leur compatibilité avec le critère de vérifiabilité doit être examinée avec discernement, ce à quoi nous nous attachons à présent.

Types de base

Les types de base (booléens, nombres...) et les fonctions associées ne posent guère de problèmes, hormis la multiplicité de choix possibles concernant les représentations arithmétiques : intervalles de valeurs, précision... Les formalismes existants divergent sur la question des types de base : la plupart fournissent des types *prédéfinis*, tandis que d'autres (par exemple, LOTOS et, dans une moindre mesure, μCRL) offrent des bibliothèques normalisées que l'utilisateur doit explicitement importer s'il le souhaite, mais qu'il peut aussi délaisser au profit d'autres bibliothèques écrites par ses soins. La première approche est simple et évite les redéfinitions (parfois incohérentes) des types de base par les utilisateurs. La seconde approche n'est pas toujours ergonomique — notamment lorsque l'utilisateur ne bénéficie pas des notations usuelles (nombres, chaînes de caractères) — mais elle possède plusieurs avantages importants : elle résout élégamment les problèmes de représentations arithmétiques, puisque l'on peut avoir autant de bibliothèques numériques que nécessaire ; elle permet d'adapter facilement le formalisme à de nouveaux domaines d'application (par exemple, le matériel) en le complétant par des bibliothèques propres à ce domaine ; enfin, elle est compatible avec les techniques d'interprétation abstraite [COU 92] puisque l'on peut abstraire un type en le remplaçant par un autre — par exemple, un type entier infini par un type fini (intervalle ou énuméré) — sans qu'il y ait besoin de modifier les définitions de processus qui utilisent ce type.

Types enregistrements

Pour les types enregistrements («**record**» en ADA, «**struct**» en C), il y a deux points délicats : les problèmes d'allocation/désallocation qui seront étudiés plus loin à propos des types dynamiques, et les problèmes liés à l'initialisation correcte des champs d'enregistrement, que nous analysons ici. Considérons l'exemple d'un type `POINT` qui est un enregistrement avec deux champs x et y de type entier.

L'approche fonctionnelle ne pose pas de problème : on crée une valeur de type `POINT` en appelant un constructeur à qui l'on passe la valeur des champs, par exemple «`POINT`$(0, 0)$» ou «`POINT`$(x$=>$0, y$=>$0)$» en E-LOTOS ; on ne modifie pas directement un enregistrement existant, mais on peut en créer une copie modifiée, par exemple «`POINT`$(p.x, p.y + 1)$» en E-LOTOS ou «$p.\{y$:=$y + 1\}$» en LOTOS NT.

L'approche impérative permet, en général, d'affecter les enregistrements, soit globalement, soit champ par champ : dans ce dernier cas, on risque d'avoir des enregistrements dont certains champs ne sont pas correctement initialisés. Ceci peut être évité soit en forçant l'initialisation des enregistrements au moment de leur déclaration ou de leur allocation, soit en effectuant une analyse statique du flot des données qui vérifie l'initialisation de chaque champ individuellement.

Types tableaux

Pour les types tableaux, les problèmes sont similaires à ceux des enregistrements, avec comme différence que les noms des champs sont remplacés par des indices dont la valeur peut n'être connue qu'à l'éxecution. D'une part, ceci pose le problème des débordements d'indice, pour lesquels il faut prévoir des levées d'exceptions et les traitements correspondants, ce qui peut s'avérer contraignant ; toutefois, une analyse statique du flot de données (par exemple, basée sur des intervalles ou des polyèdres) peut permettre de réduire cette contrainte en identifiant certains endroits où l'on peut prouver qu'il ne peut y avoir de débordement. D'autre part, le problème de l'initialisation correcte des tableaux admet différentes solutions selon que l'on autorise ou pas l'affectation individuelle des éléments de tableaux ; si elle est permise, on doit soit imposer l'initialisation globale des tableaux, soit convevoir une analyse statique plus fine.

Types unions

Les types unions (ou enregistrements avec un discriminant et des champs variants) sont indispensables, notamment dans les protocoles de télécommunications où les messages peuvent prendre diverses formes et différents champs. Il existe différentes solutions pour modéliser ces types, dont certaines sont clairement préférables. Les questions relatives à l'allocation/désallocation des valeurs de type unions seront détaillées plus loin.

Si l'on considère des unions dont la gestion du discriminant est entièrement à la charge de l'utilisateur (comme en C) ou dont le discriminant est directement manipulable par l'utilisateur (comme en PASCAL), il y a un double risque de violation du typage : l'utilisateur peut vouloir accéder à un champ incompatible avec la valeur du discriminant (la seule solution est alors de lever une exception, comme en ADA) ou vouloir modifier la valeur du discriminant sans changer celle des autres champs (la solution est, comme en ADA, de n'autoriser la modification du discriminant que par une affectation globale de l'enregistrement — ce qui permet de vérifier statiquement la compatibilité — et non pas indépendamment des autres champs).

Il est plus sûr de considérer des unions dont le discriminant est un champ implicite, que l'utilisateur ne peut ni consulter ni modifier directement (cas des types à constructeurs en ML et dans les langages basés sur la réécriture, approche reprise en E-LOTOS et LOTOS NT, cas des sous-classes d'une même classe dans les langages à objets...). Des mécanismes de haut niveau, tels que le filtrage (*pattern matching*) pour les langages fonctionnels et la liaison dynamique pour les langages à objets, garantissent que les accès aux champs coïncident avec la valeur du discriminant. Dans tous les cas, la modification du discriminant se fait par un appel de constructeur qui initialise également les autres champs.

Types dynamiques

Pour modéliser fidèlement certaines classes de systèmes, notamment dans les domaines du logiciel et des télécommunications, on doit pouvoir disposer de structures de données dynamiques (listes, arbres...). On peut citer par exemple les protocoles qui opèrent sur des réseaux dont la topologie évolue (déploiement, reconfiguration dynamique...), l'exemple le plus simple étant celui d'un protocole qui explore la topologie du réseau en mémorisant la liste des sites au fur et à mesure qu'il les découvre. Bien que beaucoup d'outils de vérification énumérative (*model checkers*) n'acceptent encore que des types finis, il est possible de traiter les types dynamiques, sous réserve que certains choix cruciaux soient faits judicieusement.

Si l'on considère des formalismes dans lesquels les pointeurs sont explicites (comme ADA, C et C++) ou qui possèdent la notion de référence vers un objet (comme EIFFEL et JAVA), alors on est confronté à plusieurs difficultés qui conditionnent, pour certains, l'existence d'une sémantique et, pour d'autres, l'efficacité de la vérification automatisée :

– Il faut interdire les conversions de pointeurs (sauf éventuellement vers un type parent ou, ce qui revient au même, vers une super-classe).

– Il est indispensable que la tentative de déréférencer un pointeur nul conduise à une levée d'exception (comme en ADA, EIFFEL et JAVA) plutôt qu'à un résultat indéfini (comme en C et C++).

– Si on veut modéliser exactement que la mémoire dynamique est finie, alors l'instruction d'allocation («**malloc**» en C, «**create**» en EIFFEL, «**new**» dans les autres langages) devient indéterministe : elle peut réussir en renvoyant un pointeur valide, ou bien échouer en renvoyant un pointeur nul ou en levant une exception. Ceci est difficilement compatible avec le souhait d'avoir des expressions déterministes ; il semble donc préférable de supposer par défaut que la mémoire est potentiellement infinie et, en cas de besoin, de modéliser un allocateur de mémoire finie par un processus indéterministe.

– Si l'on fournit une relation d'égalité entre pointeurs, c'est-à-dire si l'on permet de distinguer deux pointeurs ayant des valeurs différentes même si les cellules vers lesquelles ils pointent ont un contenu identique (ce qui est le cas dans la plupart des langages impératifs ou à objets), alors l'instruction d'allocation devient indéterministe, puisque deux allocations successives renvoient des valeurs de pointeurs différentes.

– Si l'on permet à l'utilisateur de modifier le contenu des cellules allouées, alors le coût en mémoire de la vérification énumérative peut devenir très élevé. En effet, pour chaque état mémorisé, on doit, d'une manière ou d'une autre, conserver le tas (*heap*), c'est-à-dire l'ensemble des cellules allouées et leur contenu, puisque chaque transition d'un état vers un autre est susceptible d'allouer de nouvelles cellules et/ou de modifier le contenu des cellules existantes. De plus, l'existence de synonymies (*aliasing*) complique l'analyse du flot des données et en réduit l'efficacité.

– Si l'on offre une instruction de désallocation explicite («**free**» en C, «**delete**» en C++) plutôt que l'emploi systématique d'un ramasse-miettes, alors on rend possible la création de références «pendantes» par suite de désallocations trop précoces. Celles-ci pourraient être évitées par des calculs supplémentaires à l'exécution, soit en levant une exception lorsque l'on tente de déréférencer une référence pendante, soit, lorsque l'on désalloue une cellule, en recherchant et en éliminant toutes les références vers cette cellule : dans les deux cas, ceci entraînerait un surcoût en mémoire et/ou en temps qui irait à l'encontre du but d'optimisation recherché (puisque la désallocation explicite est un mécanisme de bas niveau qui ne se justifie que par un gain de performance par rapport au ramasse-miettes). En outre, en vérification énumérative, la notion de désallocation explicite perd beaucoup de son sens, puisqu'il faut mémoriser un grand nombre d'états explorés : même si l'on rencontre une instruction de désallocation portant sur une cellule, on peut être amené à conserver cette cellule afin de ne pas perdre la trace d'un état antérieur.

– Dans le cas de processus communicants, si l'on autorise un processus d'envoyer une valeur pointeur à un autre processus s'exécutant en parallèle, alors on crée implicitement une mémoire commune entre processus, ce qui pose un double problème. D'une part, le concept de mémoire commune comporte de sérieux inconvénients que l'on détaillera plus loin, à tel point qu'aucune algèbre de processus n'a adopté les variables partagées comme mode de communication entre processus ; on pourrait instaurer un typage des communications pour garantir que les pointeurs restent locaux aux processus qui les ont alloués, mais ceci interdirait l'échange de données dynamiques entre processus. D'autre part, lorsque plusieurs processus concurrents veulent effectuer une instruction d'allocation, on risque de créer des entrelacements inutiles de transitions ; certaines analyses [IOS 02] ont été proposées pour éviter cette cause d'explosion d'états, mais elles restent complexes.

Au vu des problèmes redoutables posés par les pointeurs dans les langages impératifs et à objets, il nous semble préférable d'adopter une approche de plus haut niveau, opérationnelle depuis longtemps et utilisée aussi bien pour les langages fonctionnels qu'en vérification énumérative (notamment dans les outils CÆSAR [GAR 89b] et CÆSAR.ADT [GAR 90] qui ont été, à notre connaissance, les premiers à traiter les types dynamiques). Dans cette approche :

– Les pointeurs existent de manière sous-jacente, mais ne sont pas directement manipulables : l'opération de déréférencement est systématique et implicite.

– La relation d'égalité ne porte pas sur la valeur des pointeurs eux-mêmes, mais sur le contenu des cellules qu'ils référencent ; ainsi, deux listes chaînées pourront être déclarées égales mêmes si leurs cellules respectives ont été allouées séparément.

– L'opération d'allocation se fait au moyen de fonctions constructeurs qui initialisent immédiatement le contenu des cellules ; l'allocation est déterministe et réussit toujours — hypothèse d'une mémoire infinie.

– Il n'est pas permis de modifier les cellules déjà allouées ni de les désallouer explicitement.

Dans le cadre d'un langage de programmation, cette approche fonctionnelle peut entraîner un surcoût en mémoire par rapport à une approche impérative où l'utilisateur gère au mieux l'allocation et la désallocation de mémoire. Néanmoins, diverses techniques existent pour en améliorer les performances : utilisation d'un ramasse-miettes, mise en facteur des cellules identiques en les rangeant dans des tables de hachage (*sub-term sharing, canonical heap*), génération de code évoluée s'autorisant à modifier les cellules existantes lorsque cela est possible (*destructive updates*) ou à introduire des instructions de désallocation dès qu'une cellule n'est plus utile (*compile-time garbage collection*)...

En revanche, dans le cadre de la vérification énumérative, cette approche fonctionnelle est la plus efficace, pour une raison essentielle : au fur et à mesure que l'on explore de nouveaux états, le tas évolue de manière croissante monotone : de nouvelles cellules sont ajoutées, mais le contenu des cellules existantes n'est pas modifié. Ainsi, on peut avoir un seul tas commun à tous les états explorés —- et non pas un tas pour chaque état, comme dans l'approche impérative, ce qui n'est absolument pas une solution réaliste pour des systèmes complexes.

La principale conclusion de cette analyse comparative est donc l'incompatibilité forte entre, d'une part, la représentation des types dynamiques dans les langages impératifs ou à objets (pointeurs, références) et, d'autre part, les besoins de la vérification, en particulier énumérative. L'approche fonctionnelle semble donc s'imposer, au moins pour la description des types dynamiques, et ensuite, par effet de contagion, pour les autres types (enregistrements, tableaux, unions...). En effet, les langages fonctionnels constituent une solution cohérente, éprouvée, ergonomique et sûre ; en outre, ils sont le meilleur formalisme pour l'application des techniques de preuve. Ainsi, toutes les algèbres de processus récentes (E-LOTOS, LOTOS NT, FDR2) ont adopté ce choix.

3. Spécification des comportements

Outre les données, un formalisme pour les systèmes asynchrones doit permettre de décrire les processus concurrents, ainsi que leurs modes de synchronisation et de communication. C'est exactement la mission des algèbres de processus, dont nous examinons maintenant les aspects essentiels, en les comparant aux autres formalismes pour les systèmes asynchrones.

On ne considère ici que les algèbres de processus asynchrones, en omettant certaines variantes utiles qui seront peu ou pas abordées :

– les algèbres synchrones (SCCS [MIL 83], ESTEREL [BER 92]...) dans lesquels les processus concurrents sont cadencés par une horloge globale,

– les algèbres mobiles (π-calcul [MIL 92, SAN 01], *join calculus* [FOU 96]...) qui mettent l'accent sur l'évolution dynamique des processus et des canaux de communication,

– les algèbres temporisées (ATP [NIC 94], ET-LOTOS [LéO 97, LéO 98], *Timed* μCRL [GRO 97], TIMED CSP [DAV 95]...) dédiées à la modélisation du temps quantitatif et des notions associées (délais, urgence...),

– les algèbres probabilistes et stochastiques (PEPA [HIL 96], EMPA [BER 98], IMC [HER 02]...) qui permettent d'associer des probabilités et des durées moyennes aux événements.

3.1. *Choix liés à la représentation des comportements*

Comportements séquentiels

En première approximation, les comportements séquentiels peuvent être vus comme des automates qui décrivent, non des langages à reconnaître, mais les évolutions d'un système dans le temps. Ces automates sont étendus par des variables (souvent appelées *variables d'état*) dont la valeur peut être consultée (notamment pour permettre ou interdire le franchissement des transitions) et modifiée (par exemple, lors des changements d'états). Classiquement, il existe deux grandes approches pour spécifier les automates, dont s'inspirent la plupart des formalismes dédiés aux systèmes asynchrones :

– Les automates peuvent être donnés par la liste de leurs états et de leurs transitions, soit sous forme textuelle (ESTELLE, SDL/PR, *IO automata*...), soit sous forme graphique (*statecharts*, SDL/GR, UML/RT...). Dans ces formalismes, les variables peuvent être consultées/modifiées par des actions attachées aux états et/ou aux transitions.

– Les automates peuvent aussi être décrits par des *expressions régulières* formées d'actions atomiques combinées entre elles par trois opérateurs : concaténation, choix et répétition. Cette approche constitue la base des *expressions de chemins* [CAM 74], puis des algèbres de processus. Dans ce cadre, un *comportement* (*behavior expression*) est un terme algébrique ; la concaténation correspond à la composition séquentielle (ou préfixage) et la répétition s'obtient par un opérateur d'itération («*» en CSP, «**loop**» en E-LOTOS) et/ou des appels récursifs de processus (dans la plupart des algèbres). Naturellement, ces trois opérateurs primitifs doivent être complétés, notamment pour permettre la manipulation de données et pour définir des processus (c'est-à-dire donner un nom à certains comportements, de la même manière qu'on définit des fonctions pour nommer certaines expressions).

Il existe en fait une troisième approche basée sur la caractérisation d'automates par des formules logiques (dualité entre logique et automates) ; de par l'aspect déclaratif des logiques, cette approche est davantage utilisée pour l'expression de propriétés à vérifier que pour la modélisation des comportements eux-mêmes.

Si la représentation — textuelle ou graphique — des automates par leur liste d'états et de transitions peut séduire en raison de sa simplicité, elle présente néanmoins plusieurs inconvénients :

– Cette approche va à l'encontre des acquis de la programmation structurée : il s'agit d'une programmation «*par gotos*» qui conduit généralement à des descriptions peu structurées, lesquelles tendent à se dégrader dans le temps avec les corrections et évolutions apportées pendant la maintenance.

– Lorsque la complexité du système augmente, la représentation graphique souffre de limitations physiques (taille des écrans d'ordinateurs) ; de plus, les outils graphiques sont souvent chers, lents à utiliser, et pas toujours robustes. On peut d'ailleurs noter que les principaux langages de programmation sont textuels : l'engouement qui existait, dans les années 70, pour les organigrammes a progressivement disparu en même temps que s'imposait la programmation structurée.

– Cette representation ne permet pas, dans sa version de base, d'exprimer facilement la notion d'*interruption*, c'est-à-dire le fait qu'un comportement B_1 puisse, dans chacun de ses états, être interrompu par un autre comportement B_2. Plusieurs algèbres de processus (LOTOS, les versions récentes de CSP, et E-LOTOS) incluent un opérateur «B_1 `[>` B_2» pour modéliser cette notion. Certains langages graphiques (en particulier, SDL) ajoutent à leurs automates une transition spéciale partant de tous les états, mais cette construction ne permet pas d'exprimer les interruptions multiples au sein d'un même processus séquentiel — par exemple le comportement LOTOS suivant : «(B_1 `[>` B_2) `[]` (B'_1 `[>` B'_2)».

Il n'est donc pas certain que les représentations d'automates basés sur l'énumération des états et des transitions soient les mieux capables de répondre à la complexité croissante des systèmes. En revanche, elles constituent –

comme les réseaux de Petri — d'excellents modèles sémantiques internes sur lesquels peuvent opérer les algorithmes de compilation et d'analyse.

Comportements parallèles

Pour décrire des systèmes asynchrones, il faut pouvoir assembler des comportements séquentiels afin de les faire fonctionner en parallèle. Pour cela, il existe trois approches :

– Dans les langages graphiques (par exemple, SDL/GR, *statecharts*...), la mise en parallèle de comportements séquentiels est modélisée par la juxtaposition de «boîtes» correspondant à ces processus, éventuellement connectées par des traits qui représentent les chemins de communication entre processus. La simplicité de cette approche trouve sa contrepartie dans les limitations physiques sur la taille des schémas et la nécessité d'un mécanisme additionnel pour permettre la création dynamique de processus en nombre variable.

– D'autres langages (par exemple ESTELLE, les langages à objets concurrents...) s'inspirent de la manière dont les langages de programmation séquentiels effectuent le lancement de tâches parallèles (par exemple, ADA avec ses tâches, C avec la primitive *fork* du système UNIX...) : un processus P_1 peut créer un processus P_2 au moyen d'une instruction spécifique, qui n'est jamais bloquante, P_1 continuant immédiatement son exécution, en même temps que commence l'exécution de P_2. Cette approche est simple et flexible, mais essentiellement pour la création de processus : l'établissement des canaux de communication entre processus et la terminaison (synchronisée ou non) des processus sont généralement moins faciles à exprimer.

– Enfin, les algèbres de processus emploient des opérateurs de composition parallèle qui décrivent à la fois le lancement de tâches concurrentes, leur synchronisation et leur terminaison. Par exemple, CSP et LOTOS comportent un opérateur binaire «B_1 `|[`L`]|` B_2» signifiant que les comportements B_1 et B_2 doivent s'exécuter en parallèle, se synchroniser sur les ports de la liste L (et uniquement sur ces ports) et se terminer de manière synchrone. Cette approche a plusieurs avantages : elle établit une symétrie formelle entre démarrage et terminaison ; elle se prête bien aux preuves inductives sur la structure algébrique des comportements ; elle permet d'exprimer la création d'un nombre variable de tâches parallèles, soit par des appels récursifs de processus à travers l'opérateur parallèle, soit par des opérateurs itératifs dédiés («`par`» en LOTOS...). En revanche, l'utilisation d'opérateurs algébriques binaires souffre de limitations lorsqu'il s'agit de décrire des réseaux complexes de tâches concurrentes : c'est pourquoi des opérateurs n-aires — qui ont un lien immédiat avec les descriptions graphiques — ont été proposés [BOL 90, GAR 99] et intégrés à la norme E-LOTOS.

Ces trois approches sont suffisamment différentes pour qu'il soit difficile de les comparer dans l'absolu. On peut toutefois souligner trois points importants en faveur de l'approche algébrique :

– En général, les formalismes basés sur les automates communicants distinguent deux niveaux hiérarchiques : au niveau inférieur, des automates séquentiels qui, au niveau supérieur, sont composés en parallèle. Au contraire, les algèbres de processus (de même que les réseaux de Petri) permettent de combiner librement les opérateurs de composition séquentielle et parallèle ; ainsi, le comportement LOTOS «B_1 `>> (`B_2' `|||` B_2''`) >>` B_3» exprime qu'après l'exécution du comportement B_1, on doit lancer (*fork*) les deux comportements B_2' et B_2'', et qu'après leur terminaison synchrone (*join*), il faut exécuter le comportement B_3.

– Un système asynchrone complexe étant généralement obtenu par assemblage de sous-systèmes eux-mêmes décomposables, et ainsi de suite, on doit pouvoir spécifier ce qui, dans le comportement d'un sous-système, est suffisamment pertinent pour être observable au niveau supérieur. Dans ce but, les algèbres de processus offrent des moyens d'abstraction explicites pour masquer sélectivement les actions des sous-systèmes (opérateurs de restriction en CCS, d'abstraction en CSP et LOTOS, d'encapsulation en ACP et μCRL, d'abstraction et de renommage en E-LOTOS...). Les actions ainsi masquées deviennent *internes* (notées «τ» en CCS, «**i**» en LOTOS...).

– Sur un plan théorique, les algèbres de processus bénéficient d'une sémantique rigoureusement définie, ce qui n'est pas le cas de tous les langages. Les points sémantiques délicats ont été intensivement étudiés et les solutions sont désormais connues ; cette situation diffère, par exemple, de celle des langages à objets, où le mélange de concepts tels que l'héritage et la concurrence soulève des problèmes difficiles. Enfin, on doit constater que la quasi-totalité des algèbres de processus asynchrones ont adopté la *sémantique d'entrelacement* (*interleaving semantics*) — qui ramène la composition parallèle à une combinaison de composition séquentielle et de choix, et permet ainsi de traduire tout comportement en automate — de préférence à l'approche alternative dite du *vrai parallélisme* (*true concurrency*) [BER 01, 5ème partie].

Variables et flot de données

On a vu que les comportements devaient pouvoir manipuler des données. Les algèbres de processus fournissent divers opérateurs pour déclarer des variables typées, leur donner des valeurs, évaluer des expressions, spécifier des gardes booléennes qui conditionnent l'exécution des comportements… Ces opérateurs étant, le plus souvent, dérivés des langages fonctionnels et à commandes gardées [DIJ 75], leur aspect peut sembler hermétique ; c'est pourquoi les algèbres de processus récentes comme E-LOTOS et LOTOS NT leur préfèrent des constructions standards («**if**», «**case**», «**for**», «**while**»…) plus proches des langages de programmation usuels et, pour donner un caractère uniforme au langage, symétriques des constructions utilisées pour la description des fonctions. Les algèbres de processus permettent aussi d'émettre ou de recevoir une valeur sur un port de communication, ainsi que d'avoir des processus paramétrés par des variables auxquelles on donne des valeurs effectives lorsque ces processus sont appelés.

On ne saurait comprendre la sémantique des données manipulées au sein des comportements sans avoir à l'esprit deux règles essentielles.

D'une part, les consultations et modifications de données dans les comportements ne sont jamais observables depuis l'extérieur du comportement, c'est-à-dire qu'elles ne créent pas de transition dans l'automate correspondant au comportement : la seule exception est le cas des données concernées par une action d'émission ou de réception qui, elle, crée une transition dans l'automate. Par exemple, une séquence d'affectations à des variables ne créera aucune transition dans l'automate tant que l'on n'aura pas atteint une action de communication ou une action interne, seules capables de provoquer un changement d'état observable depuis l'extérieur.

D'autre part, conformément au critère de vérifiabilité, les algèbres de processus cherchent à garantir que toute variable est dûment affectée avant d'être utilisée. Déjà évoqué au sujet des fonctions, ce problème est ici plus ardu en raison de l'indéterminisme présent dans les comportements. Par exemple, le comportement «$(X_1 := 1$ [] $X_2 := 2)$; $X := X_1 + X_2$» pose problème car, selon la branche du choix que l'on a exécutée, l'une des deux variables, X_1 ou X_2, n'a pas été affectée (la situation serait identique si l'on remplaçait les affectations à X_1 et X_2 par des réceptions de valeurs et l'affectation à X par une émission de $X_1 + X_2$). Pour se prémunir contre de semblables difficultés, les algèbres de processus ont recours à diverses solutions. L'adoption d'un style fonctionnel, qui empêche de dissocier la déclaration d'une variable de son affectation et restreint strictement la portée des variables, évite beaucoup de problèmes, mais pas tous : l'exemple ci-dessus (dans sa version émission/réception) subsiste. Certaines algèbres (comme CCS, TCSP, LOTOS…) évitent ce problème en imposant des contraintes syntaxiques fortes sur l'opérande gauche de la composition séquentielle, à qui l'on interdit d'être autre chose qu'une action atomique (émission ou réception) : ceci conduit à un opérateur asymétrique (*préfixage*). D'autres algèbres (comme ACP, μCRL…) conservent un opérateur de composition séquentielle symétrique et autorisent donc la présence du choix indéterministe en partie gauche de cet opérateur ; elles prohibent néanmoins l'exemple ci-dessus en exigeant que toute variable déclarée dans un opérande du choix indéterministe soit locale à cet opérande et ne puisse être vue hors de cet opérande. Les algèbres modernes (comme E-LOTOS et LOTOS NT) ont des règles plus souples : on peut séparer les déclarations des affectations de variables ; l'opérateur de composition séquentielle est symétrique et l'on n'impose aucune restriction syntaxique sur son opérande gauche ; mais une analyse statique du flot des données permet de rejeter les comportements dont on ne sait pas garantir que toutes leurs variables sont correctement initialisées (ce problème étant indécidable, il ne peut s'agir que de conditions suffisantes — et pratiquement acceptables).

Indéterminisme

L'indéterminisme est un concept fondamental qui permet d'abstraire certains comportements complexes que l'on se refuse à analyser et à décrire en détail, quand bien même ils seraient intrinsèquement déterministes. Ainsi, l'indéterminisme est fréquemment utilisé pour modéliser des situations telles que l'allocation de ressources effectuée selon des critères inconnus, l'ordonnancement d'événements (*scheduling*) dont on ne connaît pas la stratégie, les pannes de processus, les pertes de messages…

L'introduction de l'indéterminisme dans les langages informatiques s'est d'abord faite dans la logique algorithmique [SAL 70] et les commandes gardées [DIJ 75], avant d'être reprise en CSP et dans toutes les algèbres de processus — à l'exception des algèbres synchrones qui mettent l'accent, au contraire, sur le déterminisme. Dans les algèbres asynchrones, l'indéterminisme est un constituant essentiel dont la présence est soit explicite (s'il est spécifié à l'aide d'un opérateur dédié), soit implicite (s'il survient comme conséquence de la sémantique d'autres opérateurs). Il en existe deux formes principales :

– L'*indéterminisme sur le contrôle* exprime que, parmi un ensemble de chemins d'exécution, un seul sera choisi. Il peut être introduit explicitement par l'opérateur de choix («+» en CCS, ACP et μCRL, «[]» en CSP, LOTOS et E-LOTOS) ou implicitement, à cause de la sémantique d'autres opérateurs tels que l'interruption, la composition parallèle (indéterminisme induit par la sémantique d'entrelacement), le masquage et le renommage des actions… Différentes situations de choix sont possibles : choix entre deux actions internes, choix entre une action interne et une action observable (émission ou réception), choix entre deux actions observables (émission/émission, émission/réception, réception/réception).

– L'*indéterminisme sur les données* consiste à sélectionner une valeur parmi un ensemble de valeurs possibles. Il peut être introduit explicitement par un opérateur qui affecte à une variable X une valeur quelconque de type T («**choice** $X : T$» en LOTOS, «**sum** $(X : T, \ldots)$» en μCRL, «$X :=$ **any** T» en E-LOTOS…) ou implicitement (dans beaucoup de sémantiques, la réception d'une valeur se ramène à un choix de valeur indéterministe). On ne suppose pas que les variables non initialisées prennent des valeurs indéterministes : en effet, il est préférable d'exprimer explicitement la volonté d'introduire du indéterminisme en utilisant un opérateur approprié et de pouvoir détecter statiquement les variables non initialisées.

3.2. *Choix liés aux primitives de synchronisation et de communication*

Un formalisme pour décrire les systèmes asynchrones doit permettre aux processus concurrents de communiquer et de se synchroniser. Historiquement [AND 83], les premières études du parallélisme ont eu lieu sur des systèmes d'exploitation en temps partagé : la communication entre processus concurrents s'effectuait naturellement par mémoire commune (variables partagées). Pour éviter les conflits d'accès à la mémoire, on a ajouté des moyens de synchronisation également basés sur l'emploi d'une mémoire centralisée (sémaphores, verrous). Puis, devant la difficulté d'écrire des programmes asynchrones corrects, des constructions plus structurées (moniteurs, sections critiques…) ont été développées, afin d'imposer une discipline de programmation susceptible d'éviter de nombreux problèmes de synchronisation.

Un pas décisif a été franchi avec l'invention du *rendez-vous* de CSP [HOA 78] qui fournit un mécanisme unifié pour la communication et la synchronisation, basé sur la notion de message et ne présupposant pas l'existence d'une mémoire commune entre les processus. Ce mécanisme a ensuite été repris en CCS et dans toutes les algèbres de processus, ainsi que dans plusieurs langages de programmation (notamment ADA, OCCAM…).

Dans sa forme originelle, le rendez-vous de CSP fait intervenir deux processus concurrents : l'un doit émettre une valeur que l'autre doit recevoir et mémoriser dans une variable. Le rendez-vous n'a lieu que lorsque les deux processus sont simultanément prêts : les processus doivent s'attendre mutuellement, ce qui fournit un moyen de synchronisation. Le rendez-vous est symétrique au sens où chaque processus (aussi bien l'émetteur que le récepteur) peut être contraint d'attendre. Il existe aussi des variantes asymétriques destinées à l'implémentation (notamment en ADA, ESTELLE, SDL…) où l'utilisation de files permet d'éviter l'attente de l'émetteur, mais il ne s'agit plus de rendez-vous à proprement parler. La proposition initiale de CSP a ensuite été améliorée et généralisée sur plusieurs points :

– Pour éviter que chaque processus ne mentionne explicitement le nom des autres processus avec lesquels il effectue des rendez-vous, ce qui limite fortement la réutilisation des processus hors du contexte pour lequel ils ont été définis, le concept de *porte* (ou *port*, *canal*…) a été introduit afin de nommer les connexions entre processus. Conventionnellement (en TCSP, LOTOS, E-LOTOS…), l'émission d'une valeur V sur une porte G est notée «$G \,!V$» et la réception d'une valeur mémorisée dans une variable X est notée «$G \,?X$».

– Une extension utile consiste à permettre la transmission de plusieurs valeurs lors d'un même rendez-vous, en émission («$G \,!V_1 \ldots \;!V_n$»), en réception («$G \,?X_1 \ldots \;?X_n$»), voire en émission/réception simultanée («$G \,!V_1 \ldots \;!V_m \,?X_1 \ldots \;?X_n$»).

– De nombreux formalismes (automates d'Arnold-Nivat, CCS, LOTOS…) permettent le rendez-vous sur une porte sans qu'il y ait émission ni réception de valeur : il s'agit alors de *synchronisation pure*, sans communication.

– Deux autres extensions (rendez-vous avec filtrage et négociation, rendez-vous à N processus) seront présentées plus loin.

Le rendez-vous est un mécanisme abstrait et puissant qui peut recevoir diverses implémentations, selon que les processus représentent des composants matériels ou logiciels, selon qu'ils s'exécutent sur un ou plusieurs processeurs, selon le mode de communication disponible entre processus… Nous examinons à présent les avantages du rendez-vous pour la modélisation des systèmes asynchrones, en le comparant aux mécanismes alternatifs proposés dans d'autres formalismes que les algèbres de processus.

Rendez-vous et événements

En premier lieu, le rendez-vous est parfaitement adapté à la programmation *dirigée par les événements* (*event driven*) dans laquelle le comportement d'un système est décrit par des réactions aux événements qu'il reçoit. L'envoi et la récupération des événements se modélisent naturellement par des rendez-vous, éventuellement accompagnés de valeurs.

Rendez-vous et appels de méthodes

Dans les formalismes à objets concurrents, la communication entre objets se fait par appel de méthodes distantes : un objet O_1 appelle une méthode M d'un objet O_2 en lui passant des paramètres $V_1, \ldots V_m$. L'appel de méthode peut être soit *bloquant* (ou *synchrone*) — O_1 est alors mis en attente jusqu'au retour de M qui lui transmet des résultats $X_1, \ldots X_n$ — soit *non bloquant* (ou *asynchrone*) — auquel cas, l'appel s'apparente au dépôt d'un message dans une file ou dans un multi-ensemble (*bag*), une situation étudiée plus loin.

Les objets étant représentés par des processus, le rendez-vous permet de modéliser l'appel synchrone, soit en un seul pas «$M\,!V_1 \ldots\ !V_m\,?X_1 \ldots\ ?X_n$» (si l'on considère l'exécution de la méthode comme une opération atomique) soit en deux pas successifs «$M_{appel}\,!V_1 \ldots\ !V_m\ ;\ M_{retour}\,?X_1 \ldots\ ?X_n$» (si l'on considère que l'exécution de la méthode prend un temps non négligeable).

Rendez-vous et variables partagées

Plusieurs formalismes utilisent les variables partagées pour réaliser la communication entre processus : il peut s'agir de langages de programmation (ADA 95, JAVA...), de modèles théoriques (*statecharts*, *timed automata*...) ou de langages dédiés servant d'entrée à divers outils de vérification (MURPHI [DIL 96], PROMELA, SMV [MCM 93, CLA 96], UPPAAL [AMN 01]...).

Le partage qui existe, au niveau des formalismes, entre rendez-vous et variables partagées reproduit la dualité, au niveau des systèmes distribués et parallèles, entre communication par messages (rendez-vous) et communication par mémoire commune (variables partagées). Du point de vue de l'implémentation, les deux approches offrent des performances comparables ; du point de vue de la modélisation des systèmes asynchrones, plusieurs raisons conduisent à préférer le rendez-vous :

– Les variables partagées sont un formalisme limité en ce qu'il présuppose l'existence d'une mémoire commune : s'il convient pour certaines applications intrinsèquement centralisées (*threads* JAVA, protocoles de cache pour machine multiprocesseur symétrique...), il est inadapté pour des systèmes asynchrones plus généraux où les processus sont véritablement distribués et où la communication ne peut se faire que par échange de messages (protocoles de télécommunication, architectures multiprocesseur asymétriques, agents répartis sur Internet...).

De plus, les tentatives pour réaliser une abstraction de mémoire commune sur des systèmes multiprocesseurs quelconques se heurtent à des difficultés sémantiques ; ainsi, le modèle de mémoire distribuée proposé pour JAVA est-il incorrect [PUG 99, PUG 00] et les solutions proposées pour le corriger ne semblent pas vérifier pas les propriétés usuelles de cohérence [YAN 01]. Au contraire, le mécanisme de rendez-vous possède une sémantique parfaitement définie et s'implémente aussi bien (au moins dans ses formes usuelles) en environnement centralisé que distribué.

– Il n'existe pas de sémantique unique pour les variables partagées : plusieurs modèles sont possibles selon que l'on autorise les processus concurrents à lire et/ou écrire simultanément une même variable ; ainsi, on distingue généralement quatre catégories : EREW, CREW, ERCW et CRCW, où les lettres E, C, R et W signifient respectivement *exclusive*, *concurrent*, *read* et *write* ; de même, on peut vouloir disposer d'instructions plus évoluées qui effectuent des lectures/écritures atomiques, par exemple l'incrémentation d'une variable, l'affectation d'une donnée référencée par un pointeur, les opérations dites *test and set*...

Il semble donc réducteur de figer un modèle précis de variables partagées ; un formalisme adéquat pour les systèmes asynchrones devrait pouvoir exprimer la diversité des différentes approches : c'est le cas des algèbres de processus qui — à l'aide du rendez-vous considéré comme mécanisme primitif — modélisent les différentes formes de variables partagées, de moniteurs et de sections critiques par des processus auxiliaires qui encapsulent les variables et en contrôlent l'accès ; les algèbres modernes, telles que E-LOTOS et LOTOS NT, simplifient cette approche en permettant de définir des bibliothèques de processus génériques et facilement réutilisables.

– D'un point de vue méthodologique, l'emploi des variables partagées est peu satisfaisant. D'une part, leur utilisation incontrôlée pose de fréquents problèmes de synchronisation (déjà mentionnés plus haut) qui ont conduit à l'émergence de disciplines de programmation et du concept alternatif de rendez-vous : de ce point de vue, il est assez paradoxal qu'un mécanisme souvent jugé de trop bas niveau pour les langages de programmation soit repris dans les formalismes de spécification.

D'autre part, les variables partagées constituent — lorsqu'elles sont utilisées directement sans être encapsulées dans des constructions de haut niveau, telles que les objets protégés (ADA) ou les méthodes synchronisées (JAVA) — un obstacle à la modularité, puisque les variables partagées créent des dépendances de données entre processus, que ces dépendances ne sont généralement pas documentées dans les interfaces, et que les processus existants peuvent difficilement être réutilisés pour d'autres applications.

– Enfin, on verra plus loin que les variables partagées semblent peu compatibles avec les notions de bisimulation et de congruence, qui sont à la base des techniques de vérification compositionnelles pour les algèbres de processus [GRA 96, KRI 97, CHE 96, CHE 99], techniques qui ont été implémentés dans plusieurs outils fonctionnant de manière quasi-automatique. Au contraire, les nombreux travaux sur la vérification compositionnelle de systèmes à mémoire commune (notamment l'approche *assume/guarantee* [PNU 85]) n'ont guère débouché sur des outils automatiques : l'utilisateur doit intervenir et guider les outils en fournissant des propriétés liant les variables partagées. Pour comprendre cette situation, on peut observer que, pour un processus pris isolément, les variables partagées constituent des points d'accès qui permettent à l'environnement de consulter/modifier à tout instant les données manipulées par ce processus. Cette hypothèse étant source de complexité et souvent trop générale par rapport à la réalité, il faut pouvoir mieux délimiter les endroits où l'environnement est susceptible de manipuler les variables partagées — ce qui est précisément le but des moniteurs, sections critiques, objets protégés, méthodes synchronisées... — la forme la plus précise étant constituée par le rendez-vous, lequel spécifie exactement les points où un processus peut émettre ou recevoir des données.

Rendez-vous et files

D'autres formalismes asynchrones (par exemple ESTELLE, SDL...) permettent la communication entre processus par des files infinies de messages : l'émission n'est pas bloquante (le processus émetteur dépose un message dans une file et continue son exécution) alors que la réception l'est. Lorsqu'on le compare au rendez-vous, ce mode de communication présente plusieurs inconvénients :

– Originellement destinées à la modélisation de protocoles de télécommunication (cas d'ESTELLE et SDL), les files s'appliquent mal à d'autres domaines, notamment les circuits matériels, pour lesquels la propagation instantanée des signaux électriques est plus fidèlement reflétée par le rendez-vous que par le dépôt d'un message dans une file (ceci explique l'intérêt des algèbres de processus en modélisation de circuits).

– Les files infinies permettent la communication entre processus, mais pas la synchronisation. Ainsi, dans le cas d'un processus «producteur» rapide et d'un processus «consommateur» lent reliés par une file infinie, on ne sait pas ralentir l'activité du producteur : les messages s'accumulent indéfiniment dans la file, l'espace des états accessibles devient infini, ce qui empêche d'utiliser les techniques standards de vérification énumérative. Pour surmonter ce problème, trois approches sont possibles, dont aucune n'est complètement satisfaisante :

1) Soit on considère que, dans une implantation réelle, le producteur et le consommateur auront des vitesses moyennes comparables (hypothèse d'équité) et l'on conserve le modèle des files infinies ; l'inconvénient majeur de cette approche est que, si les files sont fiables (c'est-à-dire qu'elles ne perdent pas les messages), tous les problèmes non-triviaux de vérification sont indécidables ;

2) Soit on introduit explicitement un protocole de contrôle de flux entre le producteur et le consommateur afin de forcer le premier à ralentir, ce qui complique la modélisation avec des détails de bas niveau, augmente le nombre d'états et de messages échangés, mais permet d'utiliser la vérification énumérative standard ;

3) Soit on remplace les files infinies par des files bornées, qui rendent l'émission bloquante lorsque la file est pleine ; c'est la solution adoptée dans les outils de vérification pour ESTELLE et SDL [JAR 88, ALG 91, ALG 93], mais elle pose des problèmes sémantiques difficiles lorsqu'elle est combinée avec la notion de transition atomique (l'impossibilité d'émettre oblige à défaire les transitions entamées).

– Plus généralement, il n'existe pas de modèle universel de file. Selon les caractéristiques du système à décrire, différentes sortes de files doivent être considérées :

- Les files sont-elles infinies ou bornées ?

- Si la file est bornée, que se passe-t-il quand elle est pleine : l'émetteur reste-t-il bloqué ou le message à émettre est-il détruit ?

- Les files préservent-elles l'ordre des messages qu'elles transportent (modèle *fifo*) ou peuvent-elles le modifier (modèle *bag*) ? Existe-t-il des messages avec des priorités différentes (messages urgents, par exemple) ?

- Les files peuvent-elles perdre les messages ? Si oui, a-t-on une hypothèse d'équité qui garantit, par exemple, qu'une file ne peut perdre indéfiniment les messages ? Ou bien, connaît-on une probabilité pour la perte des messages ?

- Les files peuvent-elles altérer les messages ? Dupliquer les messages ? Insérer des messages comme le ferait un intrus hostile dans un protocole cryptographique ?

- Les files peuvent-elles, dans un cadre temporisé, retarder les messages ? Si oui, connaît-on le temps de transit moyen des messages ?

- Les processus reçoivent-ils les messages dans une file d'entrée unique (cas de SDL) ou dans plusieurs files (cas d'ESTELLE) ?

- Les processus peuvent-ils inspecter le contenu d'une file sans y prélever les messages ? Peuvent-ils réorganiser eux-mêmes le contenu d'une file (comme en SDL) ?

Comme pour les variables partagées — mais peut-être plus encore, au vu des multiples possibilités — il semble regrettable de figer, dans un formalisme, une sorte particulière de file qui ne pourra convenir qu'à certaines applications. En comparaison, le rendez-vous est un mécanisme plus simple qui permet d'exprimer les diverses sortes de files comme des processus «ordinaires» que l'on insère aux endroits précis où se situent les files à modéliser. Les algèbres de processus modernes (comme E-LOTOS et LOTOS NT) permettent de constituer des bibliothèques de processus «files», lesquelles pourront être étendues au fur à mesure des nouveaux besoins.

Rendez-vous avec filtrage et négociation

Dans sa forme la plus générale, le rendez-vous ne se limite pas à la simple mise en correspondance d'une émission et d'une réception (comme c'était le cas à l'origine en CSP et CCS). En effet, certaines algèbres de processus (TCSP, LOTOS, μCRL, E-LOTOS...) permettent d'exprimer des contraintes (*filtrage*) sur les valeurs reçues à l'occasion des rendez-vous. Par exemple, le comportement LOTOS «$G\,\texttt{?}X : T\,\texttt{[}P(X)\texttt{]}\ \texttt{;}\ \ldots$» exprime que l'on n'accepte de lire sur la porte G qu'une valeur X de type T satisfaisant la condition booléenne $P(X)$; le rendez-vous est refusé aussi longtemps que l'environnement ne propose pas de valeur convenable. La condition $P(X)$ est évaluée *au moment* où se décide l'acceptation ou le refus du rendez-vous. L'effet serait différent si l'on avait placé cette condition *après* le rendez-vous : ainsi, le comportement LOTOS «$G\,\texttt{?}X : T\ \texttt{;}\ \ \texttt{[}P(X)\texttt{]}\ \texttt{->}\ldots$» qui signifie que le rendez-vous est accepté pour toute valeur X de type T, après quoi l'exécution se bloque si $P(X)$ n'est pas satisfaite.

Une autre extension utile consiste à permettre la *négociation* de valeurs : deux processus qui proposent chacun une réception sur la même porte avec des contraintes de filtrage différentes, par exemple

$$G\,\texttt{?}X_1 : T\,\texttt{[}P_1(X_1)\texttt{]}\ \texttt{;}\ \ldots\quad \texttt{||}\quad G\,\texttt{?}X_2 : T\,\texttt{[}P_2(X_2)\texttt{]}\ \texttt{;}\ \ldots$$

peuvent effectuer un rendez-vous s'il existe au moins une valeur V de type T telle que les deux contraintes $P_1(V)$ et $P_2(V)$ soient simultanément vraies ; s'il existe plusieurs valeurs V convenables, alors l'une d'elle est choisie de manière indéterministe ; après le rendez-vous, les deux processus reprennent leur exécution en donnant à leurs variables respectives X_1 et X_2 la même valeur V.

Ces mécanismes de filtrage et de négociation sont très puissants et autorisent un style de «spécification par contraintes» (*constraint-oriented style* [VIS 88]) dans lequel chaque processus exprime des contraintes locales dont la résolution est faite globalement, au moment du rendez-vous. Ils permettent également d'obtenir, sans étendre davantage le langage, des tableaux de portes [HOA 78]. Ainsi, le comportement LOTOS «$G\,\texttt{!}N\,\texttt{?}X : T\ \texttt{;}\ \ldots$» exprime que l'on n'accepte sur la porte G que des couples de valeurs (N, V), où V est de type T, ce qui permet, par exemple, à un processus de numéro N de filtrer les rendez-vous qui le concernent.

La négociation permet également de simplifier la définition sémantique des algèbres de processus : la distinction entre émission et réception n'est plus utile si l'on considère que toute émission «$G\,\texttt{!}V$» est un cas particulier de réception dans laquelle l'émetteur impose le choix d'une valeur V fixée, c'est-à-dire «$G\,\texttt{?}X : T\,\texttt{[}X\texttt{=}V\texttt{]}$» ; en effet, le rendez-vous «ordinaire» émetteur-récepteur s'obtient comme cas particulier du mécanisme, plus général, de négociation. Certaines algèbres comme μCRL vont encore plus loin en supprimant, au niveau syntaxique, toute différence entre émission et réception ; ceci nous semble une erreur sur le plan pratique, car la distinction entre entrées et sorties améliore la lisibilité et est indispensable pour la génération de code et la génération de tests.

Rendez-vous à N processus

Les mécanismes de communication examinés ci-dessus sont essentiellement binaires : ils font intervenir deux processus (un émetteur et un récepteur, un appelant et un appelé...). Même si la situation est parfois plus complexe

— puisque plusieurs processus peuvent appeler la même méthode, déposer un message dans la même file… — et peut donner l'impression d'un cadre plus général (N émetteurs et un récepteur, N appelants et un appelé…), il ne s'agit néanmoins que de communications binaires et non pas d'une communication simultanée entre N processus.

Plusieurs formalismes (automates d'Arnold-Nivat, algèbres de processus MEIJE, TCSP, LOTOS, E-LOTOS…) permettent d'effectuer des rendez-vous entre N processus [JOU 96]. La sémantique dérive directement du concept de négociation étendu de 2 à N processus : le rendez-vous n'a lieu que lorsque tous les processus sont prêts à y participer et, dans le cas d'un échange de valeurs, que si l'ensemble des contraintes posées par les processus admet au moins une solution (ainsi, la composition parallèle de N processus se traduit par la conjonction de N contraintes logiques).

Dans les formalismes qui imposent une distinction stricte entre émissions et réceptions (par exemple, CCS, *IO automata*…), le rendez-vous à N devient problématique et n'est, en général, pas permis : en effet, dès que l'on synchronise plus de deux processus à l'aide d'un opérateur de parallélisme que l'on souhaite commutatif et associatif, on doit définir ce que signifie la synchronisation de deux émissions, ou de deux réceptions (ce que la sémantique par négociation fait parfaitement).

En pratique, le rendez-vous à N s'avère irremplaçable lorsqu'il s'agit de modéliser certains systèmes de manière abstraite, concise et correcte. En voici quelques exemples :

– Il permet de modéliser la *diffusion*, c'est-à-dire l'envoi d'un message par un processus et sa réception simultanée par N autres processus. La diffusion (qui existe aussi dans l'algèbre synchrone ESTEREL) s'obtient simplement comme un cas particulier de négociation en synchronisant une émission avec N réceptions dépourvues de contraintes booléennes.

– Il permet de modéliser le vote à l'unanimité. Par exemple, soit une mémoire parallèle associative constituée de N cellules contenant chacune une donnée V_i distincte. Les cellules sont des processus concurrents et l'on cherche à savoir si l'une d'elle contient une valeur V donnée. Pour cela, chaque cellule compare la valeur V (qu'elle a reçue par une diffusion) avec sa propre valeur V_i, puis émet «`OK !`i» en cas d'égalité ou «`NOK`» sinon. Les N émissions «`NOK`» doivent être synchronisées (pour que le rendez-vous n'ait lieu que si V ne se trouve dans aucune cellule), tandis que les émissions «`OK !`i» ne doivent pas l'être (pour que la cellule qui possède la donnée recherchée puisse le signaler indépendamment des autres cellules).

– Il permet de modéliser des votes plus complexes qu'à l'unanimité. Ainsi, [GAR 02a] modélise en LOTOS le protocole d'arbitrage du bus SCSI-2 qui, à chaque cycle, attribue le bus au périphérique demandeur possédant le plus grand numéro SCSI ; grâce à la négociation de valeurs, chaque cycle peut être décrit par un seul rendez-vous à huit processus : on évite ainsi l'explosion combinatoire que l'on aurait avec tout formalisme limité à la communication binaire.

– Il permet de modéliser des comportements complexes comme une composition parallèle de comportements plus simples. Par exemple, le contrôleur d'un robot qui évolue dans un espace tridimensionnel peut s'exprimer par la mise en parallèle de trois contrôleurs simples, chacun surveillant le déplacement du robot sur un axe donné. Pour spécifier que l'on attend que le robot ait atteint un point (x, y, z) précis, on peut utiliser un rendez-vous à trois, ce qui évite de considérer tous les entrelacements possibles selon l'ordre des axes dans lequel le robot atteint son point d'arrivée.

– Enfin, le rendez-vous à N permet d'ajouter à un système existant des processus en parallèle, soit pour observer son comportement sans le perturber (par exemple, afin de vérifier des propriétés), soit pour le restreindre (par exemple, pour modéliser les contraintes imposées par l'environnement).

En contrepartie de son expressivité et de son haut niveau d'abstraction, le rendez-vous semble, dans sa forme la plus générale, plus difficile à implémenter que le rendez-vous originel de CSP. C'est certainement le cas pour les implémentations distribuées qui, si l'on ne dispose que de communications binaires, nécessitent des protocoles complexes pour gérer la synchronisation à N, le filtrage et la négociation. En revanche, l'implémentation semble plus facile si tous les processus s'exécutent sur la même machine ou encore s'ils sont implantés sous forme de circuits.

3.3. *Choix liés aux modèles sémantiques*

La plupart des formalismes pour les systèmes asynchrones ont une sémantique définie par traduction vers des modèles basés sur le concept d'automate. On distingue trois types de modèles sémantiques :

– Dans les modèles *à états* (appelés *structures de Kripke*), des informations sont attachées aux états uniquement ;

– Dans les modèles *à actions* (appelés *systèmes de transitions étiquetées*), des informations sont attachées aux transitions uniquement ;

– Dans les modèles *mixtes*, des informations sont attachées aux états et aux transitions ; en pratique, les modèles mixtes sont peu utilisés dans les outils de vérification.

En théorie, le choix entre modèles à états et modèles à actions peut sembler indifférent, compte tenu des résultats classiques sur la dualité entre automates de Moore et automates de Mealy, et sur les correspondances entre structures de Kripke et systèmes de transitions étiquetées [NIC 90]. Mais en pratique, lorsque l'on veut vérifier automatiquement des modèles produits à partir d'un formalisme de haut niveau, ce choix n'est plus neutre. En effet, les informations attachées aux états et aux transitions ne sont pas identiques :

– Les états du modèle mémorisent, selon le formalisme considéré, l'architecture dynamique du système (ensemble des processus existants et des interconnexions établies entre ces processus), la position des points de contrôle (compteurs programmes locaux à chaque processus), la valeur des variables d'état (locales aux processus ou partagées), les messages en transit dans les files, le tas mémoire contenant les cellules allouées dynamiquement, le temps écoulé...

– Les transitions correspondent aux changements d'état observables du système. Les informations attachées aux transitions dépendent du formalisme considéré et de son mode de communication entre processus. Avec les formalismes à variables partagées, les transitions ne portent généralement aucune information. Avec les algèbres asynchrones, chaque transition correspond à une action de communication (émission, réception, rendez-vous, action interne...) et est étiquetée par les informations relatives à cette action (porte et valeurs transmises) ; la sémantique d'entrelacement fait que deux actions ne peuvent avoir lieu simultanément. Avec les algèbres synchrones, cette hypothèse n'est plus vraie : chaque transition est donc étiquetée par un produit d'actions.

Du point de vue de la vérification automatisée, il nous semble que les modèles à actions présentent plusieurs avantages importants, que nous allons détailler.

Niveau d'abstraction des propriétés

Il faut noter d'emblée que les modèles à états et les modèles à actions correspondent à deux niveaux d'abstraction distincts :

– Les modèles à états donnent du système asynchrone une vision en *boîte blanche/boîte grise*, puisque l'on peut observer tous les détails locaux à chaque processus, ainsi qu'on le ferait avec un outil de mise au point interactive (*debugger*) ;

– Les modèles à actions fournissent au contraire une vision en *boîte noire* du système, car seules les interactions du système avec son environnement — c'est-à-dire l'interface qu'il offre au monde extérieur — sont observables ; une double opération d'abstraction est appliquée, d'une part, en retirant toute information des états, et d'autre part, en renommant en action interne toute action qui ne correspond pas à une interaction extérieure du système.

Selon le type du modèle, les propriétés à vérifier s'expriment différemment. Les modèles à états conduisent naturellement à l'écriture de prédicats portant sur les variables d'états, et notamment d'*invariants*. Bien que largement répandue, cette approche pose de sérieux problèmes :

– D'une part, les variables d'états présentes dans la modélisation formelle d'un système ne sont, en général, pas propres au système lui-même, mais résultent davantage du choix personnel de l'individu qui a produit la modélisation : un autre individu en charge de modéliser ce même système aurait probablement opté pour un ensemble de variables différent. L'utilisation d'invariants comporte trois inconvénients :

- le risque de biaiser la vérification, puisque les invariants n'expriment pas tant les propriétés du système lui-même que celles d'une modélisation particulière,

- l'impossibilité d'exprimer les propriétés à vérifier tant que la modélisation du système n'est pas disponible, ce qui complique la répartition des tâches au sein d'une équipe,

- l'obligation de mettre à jour les propriétés lorsque la modélisation évolue (ajout ou renommage de variables, par exemple).

– D'autre part, les outils de vérification énumérative, lorsqu'ils construisent le modèle correspondant à un système asynchrone, effectuent souvent diverses optimisations destinées à lutter contre l'explosion d'états. En particulier, ils peuvent supprimer les variables dont la valeur est constante, décider de remettre à zéro une variable dès que sa valeur n'est plus utile, de fusionner des variables distinctes en les allouant dans le même emplacement mémoire… La présence d'invariants oblige à désactiver ces optimisations, au moins pour les variables présentes dans les invariants, afin que leur valeur ne soit pas subrepticement modifiée. Ceci réduit l'efficacité de la vérification.

– Enfin, il n'est pas toujours aisé, dans les invariants, de nommer les variables de manière univoque ; c'est notamment le cas pour les variables locales à un processus lorsque plusieurs instances de ce processus s'exécutent simultanément ; le problème se complique si les processus sont créés dynamiquement et reçoivent donc des noms générés automatiquement.

C'est pour éviter ces difficultés — et compte tenu du fait qu'en style fonctionnel, la notion de variable n'existe pas véritablement — que les algèbres de processus utilisent des modèles à actions plutôt qu'à états. Ce choix conduit à exprimer les propriétés de manière plus abstraite, en s'intéressant aux interfaces du système plutôt qu'à ses détails internes. Il est possible, avec les algèbres de processus, d'exprimer des invariants, mais ceux-ci doivent être «locaux» (c'est-à-dire ne porter que sur les variables d'un processus séquentiel) et directement insérés dans le code du processus concerné (afin de pouvoir nommer les variables et accéder à leur valeur).

Nous poursuivons à présent la comparaison des modèles à états et des modèles à actions, en mettant l'accent sur les besoins respectifs des deux grandes approches pour la vérification, celle où l'on vérifie des propriétés logiques sur un modèle (*model checking*) et celle où l'on compare deux modèles au moyen de relations d'équivalence ou de préordre (*equivalence checking*).

Vérification de propriétés logiques

En premier lieu, il faut noter que le type de logique utilisable dépend du modèle considéré : avec un modèle à états, on doit utiliser une logique dédiée aux structures de Kripke (invariants, LTL, CTL, μ-calcul propositionnel…) ; avec un modèle à actions, on doit utiliser une logique opérant sur les systèmes de transitions étiquetées (expressions régulières, PDL, ALTL, ACTL, μ-calcul modal…). En revanche, la distinction entre modèles à états et modèles à actions est parfaitement orthogonale à la distinction classique entre logiques du temps linéaire et logiques du temps arborescent. Ainsi :

– l'outil SPIN [HOL 91, HOL 97] utilise une logique linéaire (LTL) pour vérifier des modèles à états ;

– l'outil SMV [MCM 93, CLA 96] utilise une logique arborescente (CTL) pour vérifier des modèles à états ;

– l'outil TRACTA [GIA 99] utilise une logique linéaire (ALTL) pour vérifier des modèles à actions et l'outil FDR2 [ROS 97] utilise une sémantique de traces (linéaire) pour vérifier des modèles à actions ;

– la boîte à outils CADP [GAR 02b] utilise des logiques arborescentes pour vérifier des modèles à actions.

En pratique, il est rare que l'on n'ait qu'une seule propriété logique à vérifier sur un modèle donné : on doit plutôt évaluer un ensemble de plusieurs propriétés (l'approche consistant à rassembler artificiellement différentes propriétés en une seule, au moyen de conjonctions, n'est généralement pas réaliste, puisque l'on a souvent besoin de diagnostics expliquant pourquoi l'une des propriétés est fausse). En supposant que le modèle n'est pas de trop grande taille et que l'on peut le construire entièrement, on a alors le choix entre deux approches :

– Soit on effectue une vérification *à la volée*, en construisant le modèle en même temps que l'on évalue une propriété. Ceci oblige à construire le modèle autant de fois qu'il y a de propriétés à vérifier, d'où un surcoût qui peut être limité lorsque l'on sait détecter rapidement qu'une propriété est fausse sans explorer la totalité du modèle, ou lorsque l'on sait abstraire le modèle en fonction de la propriété à vérifier (τ-confluence, *slicing*…). Aussi bien les modèles à états que les modèles à transitions permettent la vérification à la volée.

– Soit on ne construit le modèle qu'une seule fois avant de vérifier successivement chaque propriété sur ce modèle. La mémoire disponible étant limitée, les phases de construction du modèle et d'évaluation des propriétés sont séparées et opèrent différemment. Pendant la construction, les transitions sont écrites immédiatement sur disque (dans un fichier), tandis que les états doivent être conservés en mémoire afin de détecter les circuits. Dans le cas d'un modèle à états, les informations sur les états doivent également être recopiées sur disque en fin de génération, puisque la phase de vérification aura besoin de ces informations pour évaluer les propriétés logiques portant sur les variables d'états. Dans le cas d'un modèle à actions, cette recopie n'est pas nécessaire, puisque les propriétés ne portent que sur les transitions. Pour un modèle à états avec n états, m transitions, s bits d'information attachés à chaque état et t bits d'information utilisés pour représenter une transition d'un état vers un autre, la taille du fichier correspondant est $T_e = ns + mt$. Pour un modèle à actions avec n états, m transitions, l bits d'information attachés à chaque transition et t bits d'information utilisés pour représenter une transition d'un état vers un autre (en sus de l'information déjà exprimée par l), la taille du fichier correspondant est $T_a = m(t+l)$. Après introduction du facteur de branchement moyen $\delta = m/n$, la comparaison $T_a < T_e$ se ramène à $\delta l < s$. En analysant les caractéristiques de nombreux «gros» modèles produits à partir d'algèbres de processus (LOTOS et μCRL), on constate que le facteur de branchement reste faible ($\delta < 10$) et, qu'en pratique, le nombre L d'étiquettes ne dépasse pas quelques centaines ($L < 1000 \Longrightarrow l = \log_2 L < 10$) : sous ces hypothèses, le modèle à actions conduit à des fichiers plus compacts que le modèle à états dès que $s > 100$ bits (soit 13 octets), ce qui est généralement le cas (les informations attachées aux états occupant souvent plusieurs dizaines, voire centaines, d'octets). Même si des techniques de compression de fichiers peuvent être appliquées (notamment pour réduire la valeur de t, comme c'est le cas dans le format BCG de la boîte à outils CADP [GAR 02b]), elles ne semblent pas susceptibles d'inverser cette situation.

Il apparaît donc que les modèles à actions sont plus polyvalents, au sens où ils conviennent à d'autres approches que la vérification à la volée.

Vérification d'équivalences et de préordres

De manière alternative (ou complémentaire) à l'usage de propriétés logiques, on peut vérifier des systèmes au moyen de relations d'équivalence ou de préordre entre modèles. Parmi les relations les plus fréquemment employées figurent les *bisimulations* [PAR 81], dont il existe de multiples variantes [GLA 01]. Ces relations peuvent être utilisées de deux manières principales :

– soit pour *comparer* deux modèles afin de décider s'ils sont équivalents ou si l'un des deux est inclus dans l'autre au sens d'un préordre ; cette approche permet de démontrer l'équivalence de comportements ou le raffinement d'un comportement par un autre ;

– soit pour *minimiser* un modèle selon une relation d'équivalence choisie, ce qui consiste, pour les bisimulations, à fusionner les états dont les comportements futurs potentiels sont équivalents ; cette approche correspond à une abstraction puisque l'on remplace un modèle par un modèle équivalent, mais comportant moins d'états.

La théorie des bisimulations est riche de résultats profonds et utiles. Ainsi, la *compatibilité* (ou *adéquation*) entre équivalences et logiques temporelles permet de remplacer un modèle par un modèle équivalent (en particulier, minimal) tout en préservant la validité de (certaines classes de) propriétés logiques évaluées sur ce modèle. De plus, il existe des algorithmes efficaces qui font que, contrairement à d'autres techniques d'abstraction, l'emploi des bisimulations peut être entièrement automatisé.

En principe, les bisimulations sont applicables aussi bien aux modèles a états qu'aux modèles à transitions. Ainsi, l'algorithme de minimisation de Paige-Tarjan [PAI 87] a été initialement conçu pour des structures de Kripke avant d'être généralisé ensuite [FER 90]. De même, les résultats de compatibilité avec les logiques sont indépendants du type de modèle considéré.

En pratique toutefois, force est de constater que la quasi-totalité des outils de bisimulation opèrent sur des modèles à actions. On peut avancer plusieurs explications à ce phénomène :

– La présence d'informations attachées aux états gêne les outils de bisimulation dont le travail consiste essentiellement à fusionner des états différents en un même état. Si l'on conserve systématiquement toutes les informations des états, alors les possibilités de minimisation sont réduites : il faut donc permettre à l'utilisateur d'indiquer les informations qu'il souhaite préserver et celles qui peuvent être abstraites, ce qui fait perdre à la vérification son caractère automatique.

– Il existe des liens étroits entre la théorie des bisimulations et celles des algèbres de processus. Ainsi, l'équivalence de deux comportements, considérés comme des termes, peut être démontrée en utilisant les axiomes de la sémantique algébrique ou par induction sur les règles de la sémantique opérationnelle. De même, les bisimulations possèdent une propriété de congruence vis-à-vis des opérateurs de composition parallèle et d'abstraction (de la plupart des) algèbres de processus : cette propriété autorise la *minimisation compositionnelle* en permettant de remplacer, dans un réseau de processus communicants, un processus par un autre équivalent (généralement obtenu par minimisation). Il semble donc raisonnable, en concevant un outil de bisimulation, de veiller à ce qu'il puisse être largement utilisé et, par conséquent, applicable aux algèbres de processus. Or les modèles à états ne conviennent pas pour les algèbres de processus, à cause du caractère fonctionnel de ces dernières. Par exemple, les deux comportements LOTOS «G **!**0 **;stop**» et «**let** X :*Nat*=0 **in** G **!** X **;stop**» sont équivalents au sens de la bisimulation, mais leurs modèles risquent de ne pas l'être si les états du second modèle font référence à la variable X.

Ici encore, il semble que les modèles à actions soient, d'un point de vue pratique, plus simples et mieux adaptés à la vérification automatisée.

4. Conclusion

Dans cet article, nous avons tenté d'expliquer pourquoi les multiples formalismes proposés pour la modélisation des systèmes asynchrones ne sont pas tous égaux, et pourquoi les algèbres de processus — combinées avec des types de données fonctionnels — fournissent un bon compromis entre quatre exigences souvent contradictoires :

– l'expressivité, qui résulte de la possibilité de combiner librement un nombre (volontairement) réduit de concepts primitifs judicieusement choisis ;

– l'universalité, qui découle du caractère mathématique des algèbres de processus qui les rend indépendantes de tout secteur d'activité particulier, comme en témoignent plusieurs centaines d'applications dans des domaines très divers ;

– l'exécutabilité, qui est attestée par l'existence de nombreux outils (CADP [GAR 02b], CWB-NC [CLE 02], FDR2 [ROS 97], LTSA [MAG 99a], μCRL *toolset* [GRO 01]...) destinés à la simulation ou à la génération d'implantations (centralisées, distribuées ou sous forme de circuits) ; en outre, il existe des algorithmes pour traduire les algèbres de processus vers des modèles de plus bas niveau tels que réseaux de Petri [GAR 89a, TAU 89, OLD 91, BES 01] et automates communicants [DUB 89, KAR 92].

– la vérifiabilité, que favorise l'adéquation des algèbres de processus aux techniques de vérification énumérative (vérification de propriétés logiques, vérification d'équivalences) aussi bien qu'aux techniques de preuve (équivalences de programmes, raffinements). On peut l'expliquer par de multiples facteurs : existence de plusieurs sémantiques formelles bien étudiées (axiomatique, opérationnelle), nombre limité de concepts qui réduit la complexité des outils, abstractions directement incorporées à la sémantique des algèbres de processus, résultats théoriques forts liant équivalences et propriétés logiques, équivalences et opérateurs de composition parallèle et d'abstraction, permettant ainsi la vérification compositionnelle...

On doit cependant constater qu'en dépit de ces qualités, la diffusion des algèbres de processus dans l'industrie existe, mais reste limitée — ce qui est généralement le cas des méthodes formelles. Le principal obstacle à une diffusion accrue est l'apprentissage et la maîtrise des algèbres de processus, qui exigent des efforts et du temps. Ceci peut s'expliquer par deux raisons objectives :

– la difficulté intrinsèque des concepts sous-jacents aux systèmes asynchrones (indéterminisme, parallélisme, synchronisation, communication...) ; de ce point de vue, les algèbres de processus sont idéales pour l'enseignement, car chaque concept y est représenté par un opérateur syntaxiquement différent, ce qui contribue à donner des idées claires aux étudiants ;

– l'inévitable remise en question des acquis antérieurs, dans la mesure où l'on inculque habituellement une vision séquentielle aux concepteurs de logiciels et une vision synchrone/déterministe aux concepteurs de matériels ; le passage au monde asynchrone/indéterministe nécessite sinon de renoncer à ces acquis, du moins à les replacer en perspective.

Toutefois, la difficulté d'apprentissage provient également de facteurs humains :

– Les algèbres de processus sont souvent pénalisées par leur syntaxe ésotérique, éloignée des langages de programmation usuels, due au fait que leurs auteurs ont fait passer les problèmes théoriques avant les questions d'ergonomie. De même, des résultats fondamentaux sont souvent, malgré leur portée générale, dissimulés sous une formalisation à outrance.

– La communauté des algèbres de processus n'a pas su se regrouper autour d'un formalisme unique, qui lui aurait permis d'atteindre une masse critique suffisante pour l'enseignement, le développement d'outils et les applications. L'opportunité offerte en 1989 par la normalisation de LOTOS n'a pas été saisie autant qu'elle aurait dû l'être : beaucoup d'équipes l'ont adopté, mais d'autres ont préféré poursuivre leurs travaux avec les formalismes existants (CCS, CIRCAL, CSP...) ou même en créer de nouveaux, similaires mais incompatibles (PSF, μCRL...).

De fait, les algèbres de processus sont — comme les méthodes formelles en général — concurrencées par deux approches alternatives :

– Les méthodes *semi-formelles*, dont la diffusion est favorisée par des notations graphiques attrayantes et un faible niveau de contraintes sémantiques ; elles servent principalement à la production de documentation et la génération de code ; leur utilisation pour la vérification automatisée reste expérimentale, leur emploi pour la preuve semblant hors d'atteinte.

– Les techniques de vérification opérant directement au niveau de programmes C, JAVA... (*software model checking*) plutôt que sur des modélisations formelles. Ces techniques sont généralement bien accueillies parce qu'elles ne remettent pas en question les méthodes de développement existantes, et parce qu'elles se présentent comme une alternative à l'emploi de méthodes formelles. Même si leur utilisation améliore significativement la qualité de «petits» programmes séquentiels, voire parallèles, il n'est pas certain qu'elle se généralise à des applications critiques de plus grande taille : en effet, les langages de programmation actuels sont de trop bas niveau (notamment pour le parallélisme) et n'ont pas été prévus pour la vérification compositionnelle.

On peut être donc être raisonnablement optimiste sur l'avenir des algèbres de processus, au vu des défis à relever par l'industrie :

– *Complexification croissante des applications* : les algèbres de processus n'ont pas les limitations des formalismes graphiques concernant le passage à l'échelle ; par ailleurs, un rapprochement entre formalismes de modélisation et langages de programmation a été résolument entrepris avec des algèbres de processus telles que E-LOTOS et LOTOS NT.

– *Besoins de sécurité accrus* : les algèbres de processus ont été expressément conçues pour la vérification et la preuve ; en particulier, leur sémantique du parallélisme rend possible la vérification compositionnelle afin de lutter contre l'explosion d'états qui reste un défi majeur.

A cet égard, on doit noter la récente percée des logiques temporelles parmi les concepteurs de circuits, qui démontre que, dès que le besoin de qualité se fait suffisamment sentir, l'industrie est capable d'adopter de nouveaux formalismes — aussi abstraits, sinon plus, que les algèbres de processus — pourvu qu'ils améliorent significativement les méthodes de développement.

Remerciements

L'auteur tient à remercier David Champelovier, Nicolas Descoubes, Frédéric Lang, Radu Mateescu, Solofo Ramangalahy et Wendelin Serwe pour leurs commentaires judicieux concernant cet article.

5. Bibliographie

[ABR 96] ABRIAL J. R., *The B Book*, Cambridge University Press, octobre 1996.

[ACE 01] ACETO L., FOKKINK W., VERHOEF C., « Structural Operational Semantics », BERGSTRA J., PONSE A., SMOLKA S., Eds., *Handbook of Process Algebra*, chapitre 3, p. 197–292, North-Holland, 2001.

[ALG 91] ALGAYRES B., COELHO V., DOLDI L., GARAVEL H., LEJEUNE Y., RODRÍGUEZ C., « VESAR : Un Outil pour la Spécification et la Vérification Formelle de Protocoles », RAFIQ O., Ed., *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'91 (Pau, France)*, Paris, septembre 1991, Hermès, p. 253–276.

[ALG 93] ALGAYRES B., LEJEUNE Y., HUGONNET F., HANTZ P., « The AVALON projet : A VALidatiON Environment For SDL/MSC Descriptions », *6th SDL forum (Darmstadt, Germany)*, 1993.

[AMN 01] AMNELL T., BEHRMANN G., BENGTSSON J., D'ARGENIO P. R., DAVID A., FEHNKER A., HUNE T., JEANNET B., LARSEN K. G., MÖLLER M. O., PETTERSSON P., WEISE C., YI W., « UPPAAL - Now, Next, and Future », CASSEZ F., JARD C., ROZOY B., RYAN M., Eds., *Modelling and Verification of Parallel Processes*, n° 2067 Lecture Notes in Computer Science, Springer Verlag, 2001, p. 100–125.

[AND 83] ANDRÉ F., HERMAN D., VERJUS J. P., *Synchronisation des programmes parallèles*, Dunod, 1983.

[ARM 93] ARMSTRONG J., VIRDING R., WILLIAM M., *Concurrent Programming in Erlang*, Prentice Hall, 1993.

[ARN 92] ARNOLD A., *Systèmes de transitions finis et sémantique des processus communicants*, Masson, 1992.

[BER 84] BERGSTRA J. A., KLOP J. W., « Process Algebra for Synchronous Communication », *Information and Computation*, vol. 60, 1984, p. 109–137.

[BER 85] BERGSTRA J. A., KLOP J. W., « Algebra of Communicating Processes with Abstraction », *Theoretical Computer Science*, vol. 37, 1985, p. 77–121.

[BER 92] BERRY G., GONTHIER G., « The Esterel Synchronous Programming Language : Design, Semantics, Implementation », *Science of Computer Programming*, vol. 19, n° 2, 1992, p. 87–152.

[BER 94] BERTHOMIEU B., LE SERGENT T., « Programming with behaviors in an ML framework, the syntax and semantics of LCS », *European Symposium On Programming (Edinburgh, UK)*, vol. 788 de *Lecture Notes in Computer Science*, Springer Verlag, avril 1994.

[BER 95] BERTHOMIEU B., « Process Calculi at work – An account of the LCS project », *Workshop on Parallel Symbolic Languages and Systems (Beaune, France)*, vol. 1068 de *Lecture Notes in Computer Science*, Springer Verlag, octobre 1995.

[BER 96] BERTHOMIEU B., « CCS Programming in an ML Framework : An account of LCS », NIELSON F., Ed., *ML with Concurrency, Design, Analysis, Implementation, and Application*, Monographs in Computer Science, Springer Verlag, 1996.

[BER 98] BERNARDO M., GORRIERI R., « A Tutorial on EMPA : A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time », *Theoretical Computer Science*, vol. 202, 1998, p. 1–54.

[BER 01] BERGSTRA J., PONSE A., SMOLKA S., Eds., *Handbook of Process Algebra*, North-Holland, 2001.

[BES 01] BEST E., DEVILLERS R., KOUTNY M., « A Unified Model for Nets and Process Algebras », BERGSTRA J., PONSE A., SMOLKA S., Eds., *Handbook of Process Algebra*, chapitre 14, p. 873–944, North-Holland, 2001.

[BOL 90] BOLOGNESI T., « A Graphical Composition Theorem for Networks of Lotos Processes », SOCIETY I. C., Ed., *Proceedings of the 10th International Conference on Distributed Computing Systems, Washington, USA*, IEEE, mai 1990, p. 88–95.

[BRO 84] BROOKES S. D., HOARE C. A. R., ROSCOE A. W., « A Theory of Communicating Sequential Processes », *Journal of the ACM*, vol. 31, n° 3, 1984, p. 560–599.

[CAM 74] CAMPBELL R., HABERMANN A., « The Specification of Process Synchronization by Path Expressions », vol. 16 de *Lecture Notes in Computer Science*, Springer Verlag, 1974, p. 89-102.

[CHE 96] CHEUNG S. C., KRAMER J., « Context Constraints for Compositional Reachability », *ACM Transactions on Software Engineering Methodology TOSEM*, vol. 5, n° 4, 1996, p. 334–377.

[CHE 99] CHEUNG S. C., KRAMER J., « Checking Safety Properties Using Compositional Reachability Analysis », *ACM Transactions on Software Engineering and Methodology*, vol. 8, n° 1, 1999, p. 49–78.

[CLA 96] CLARKE E. M., MCMILLAN K. L., CAMPOS S., HARTONAS-GARMHAUSEN V., « Symbolic model checking », ALUR R., HENZINGER T. A., Eds., *Proceedings of the 8th International Conference on Computer Aided Verification CAV*, vol. 1102 de *Lecture Notes in Computer Science*, Springer Verlag, juillet 1996, p. 419–422.

[CLE 99] CLEAVELAND R., SMOLKA S. A., « Process Algebra », WEBSTER J., Ed., *Encyclopedia of Electrical Engineering*, John Wiley and sons, avril 1999.

[CLE 02] CLEAVELAND R., SIMS S., « Generic Tools for Verifying Concurrent Systems », *Science of Computer Programming*, vol. 41, n° 1, 2002, p. 39–47.

[COU 92] COUSOT P., COUSOT R., « Abstract interpretation and application to logic programs », *Journal of Logic Programming*, vol. 13, n° 2–3, 1992, p. 103–179.

[DAV 95] DAVIES J. W., SCHNEIDER S. A., « A Brief History of Timed CS », *Theoretical Computer Science*, vol. 138, n° 2, 1995, p. 243–271.

[de 85] DE SIMONE R., « Higher level synchronizing devices in MEIJE », *Theoretical Computer Science*, vol. 37, 1985, p. 245–267.

[DIJ 75] DIJKSTRA E., « Guarded commands, non-determinancy and formal derivation of programs », *Communications of the Association of Computing Machinery*, vol. 18, 1975, p. 453–457.

[DIL 96] DILL D., « The Murφ Verification System », ALUR R., HENZINGER T., Eds., *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96*, vol. 1102 de *Lecture Notes in Computer Science*, Springer Verlag, juillet 1996, p. 390–393.

[DUB 89] DUBUIS E., « An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports », VUONG S. T., Ed., *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, North-Holland, décembre 1989.

[FER 90] FERNANDEZ J.-C., « An Implementation of an Efficient Algorithm for Bisimulation Equivalence », *Science of Computer Programming*, vol. 13, n° 2–3, 1990, p. 219–236.

[FOK 00] FOKKINK W., *Introduction to Process Algebra*, Texts in Theoretical Computer Science, Springer Verlag, 2000.

[FOU 96] FOURNET C., GONTHIER G., « The reflexive chemical abstract machine and the join-calculus », *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, janvier 1996, p. 372–385.

[GAR 89a] GARAVEL H., *Compilation et vérification de programmes LOTOS*, Thèse de Doctorat, Université Joseph Fourier (Grenoble), novembre 1989.

[GAR 89b] GARAVEL H., « Compilation of LOTOS Abstract Data Types », VUONG S. T., Ed., *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, North-Holland, décembre 1989, p. 147–162.

[GAR 90] GARAVEL H., SIFAKIS J., « Compilation and Verification of LOTOS Specifications », LOGRIPPO L., PROBERT R. L., URAL H., Eds., *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, IFIP, North-Holland, juin 1990, p. 379–394.

[GAR 97] GARAVEL H., MOUNIER L., « Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks », *Science of Computer Programming*, vol. 29, n° 1–2, 1997, p. 171–197, Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Full version available as INRIA Research Report RR-2986.

[GAR 99] GARAVEL H., SIGHIREANU M., « A Graphical Parallel Composition Operator for Process Algebras », WU J., GAO Q., CHANSON S. T., Eds., *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, IFIP, Kluwer Academic Publishers, octobre 1999, p. 185–202.

[GAR 02a] GARAVEL H., HERMANNS H., « On Combining Functional Verification and Performance Evaluation using CADP », ERIKSSON L.-H., LINDSAY P. A., Eds., *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, vol. 2391 de *Lecture Notes in Computer Science*, Springer Verlag, juillet 2002, p. 410–429, Full version available as INRIA Research Report 4492.

[GAR 02b] GARAVEL H., LANG F., MATEESCU R., « An Overview of CADP 2001 », *European Association for Software Science and Technology (EASST) Newsletter*, vol. 4, 2002, p. 13–24, Also available as INRIA Technical Report RT-0254 (December 2001).

[GIA 99] GIANNAKOPOULOU D., KRAMER J., CHEUNG S. C., « Analysing the behaviour of distributed systems using TRACTA », *Journal of Automated Software Engineering, Special issue on Automated Analysis of Software*, vol. 6, n° 1, 1999, p. 7–35.

[GLA 01] VAN GLABBEECK R., « The Linear Time–Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes », BERGSTRA J., PONSE A., SMOLKA S., Eds., *Handbook of Process Algebra*, chapitre 1, p. 3–95, North-Holland, 2001.

[GRA 96] GRAF S., STEFFEN B., LÜTTGEN G., « Compositional Minimization of Finite State Systems using Interface Specifications », *Formal Aspects of Computation*, vol. 8, n° 5, 1996, p. 607–616.

[GRO 95] GROOTE J., PONSE A., « The syntax and semantics of μCRL », PONSE A., VERHOEF C., VAN VLIJMEN S., Eds., *Algebra of Communicating Processes '94*, Workshops in Computing Series, Springer Verlag, 1995, p. 26–62, Also appeared as : Technical Report CS-R9076, CWI, Amsterdam, 1990.

[GRO 97] GROOTE J., « The syntax and semantics of timed μCRL », Technical Report n° SEN-R9709, 1997, CWI, Amsterdam.

[GRO 01] GROOTE J., LISSER B., « Computer assisted manipulation of algebraic process specifications », Technical Report n° SEN-R0117, 2001, CWI, Amsterdam.

[HAB 01] HABRIAS H., *Spécification formelle avec B*, Lavoisier-Hermès, 2001.

[HAL 93] HALBWACHS N., *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[HAR 87] HAREL D., « StateCharts : A Visual Formalism for Complex Systems », *Science of Computer Programming*, vol. 8, n° 3, 1987, p. 231–274.

[HER 02] HERMANNS H., *Interactive Markov Chains : and the Quest for Quantified Quality*, vol. 2428 de *Lecture Notes in Computer Science*, Springer Verlag, 2002.

[HIL 96] HILLSTON J., *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.

[HOA 78] HOARE C. A. R., « Communicating Sequential Processes », *Communications of the ACM*, vol. 21, n° 8, 1978, p. 666–677.

[HOA 85] HOARE C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[HOL 91] HOLZMANN G. J., *Design and Validation of Computer Protocols*, Software Series, Prentice Hall, 1991.

[HOL 97] HOLZMANN G., « The Model Checker SPIN », *IEEE Transactions on Software Engineering*, vol. 23, n° 5, 1997, p. 279–295.

[IOS 02] IOSIF R., « Symmetry Reduction Criteria for Software Model Checking », *Proc. 9th SPIN Workshop on Software Model Checking*, vol. 2318 de *Lecture Notes in Computer Science*, Springer Verlag, avril 2002, p. 22–41.

[ISO 88] ISO/IEC, « ESTELLE — A Formal Description Technique Based on an Extended State Transition Model », International Standard n° 9074, septembre 1988, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève.

[ISO 89] ISO/IEC, « LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour », International Standard n° 8807, septembre 1989, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève.

[ISO 95] ISO/IEC, « Vienna Development Method – Specification Language – Part 1 : Base Language », Internation Standard n° 13817-1 :1996, décembre 1995, International Organization for Standardization — Programming languages, their environments and system software interfaces, Genève.

[ISO 01] ISO/IEC, « Enhancements to LOTOS (E-LOTOS) », International Standard n° 15437 :2001, septembre 2001, International Organization for Standardization — Information Technology, Genève.

[ISO 02] ISO/IEC, « Z Formal Specification Notation - Syntax, Type System and Semantics », International Standard n° 13568 :2002, 2002, International Organization for Standardization — Information Technology, Genève.

[ITU 99] ITU-T, « Specification and Description Language (SDL) », ITU-T Recommendation n° Z.100, novembre 1999, International Telecommunication Union, Genève.

[JAR 88] JARD C., GROZ R., MONIN J.-F., « Development of VEDA : A Prototyping Tool for Distributed Systems », *IEEE Transactions on Software Engineering*, vol. 14, n° 3, 1988.

[JEF 95] JEFFREY A., GARAVEL H., LEDUC G., PECHEUR C., SIGHIREANU M., « Towards a proposal for datatypes in E-LOTOS », Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, octobre 1995.

[JOR 84] JORRAND P., « FP2 : Functional Parallel Programming Based on Term Substitution », BIBEL W., PETKOFF B., Eds., *Proceedings of the International Conference on Artificial Intelligence : Methodology, Systems, Applications AIMSA'84 (Varna, Bulgaria)*, North-Holland, septembre 1984, p. 95–112.

[JOU 96] JOUNG Y.-J., SMOLKA S. A., « A comprehensive study of the complexity of multiparty interaction », *Journal of the ACM*, vol. 43, n° 1, 1996, p. 75–115.

[KAR 92] KARJOTH G., « Generating Transition Graphs from LOTOS Specifications », DIAZ M., GROZ R., Eds., *Proceedings of the 5th International Conference on Formal Description Techniques FORTE'92 (Lannion, France)*, IFIP, octobre 1992, p. 275–288.

[KRI 97] KRIMM J.-P., MOUNIER L., « Compositional State Space Generation from LOTOS Programs », BRINKSMA E., Ed., *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, vol. 1217 de *Lecture Notes in Computer Science*, Berlin, avril 1997, Springer Verlag, Extended version with proofs available as Research Report VERIMAG RR97-01.

[LAM 94] LAMPORT L., « The Temporal Logic of Actions », *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 3, 1994, p. 872–923, ACM Press.

[LAM 02] LAMPORT L., *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional (Pearson Education, Inc.), 2002.

[LAN 77] LANN G. L., « Distributed Systems — Towards a Formal Approach », GILCHRIST B., Ed., *Information Processing 77*, IFIP, North-Holland, 1977, p. 155–160.

[LéO 97] LÉONARD L., LEDUC G., « An Introduction to ET-LOTOS for the Description of Time-Sensitive Systems », *Computer Networks and ISDN Systems*, vol. 29, n° 3, 1997, p. 271–292.

[LéO 98] LÉONARD L., LEDUC G., « A Formal Definition of Time in LOTOS », *Formal Aspects of Computing*, vol. 10, n° 3, 1998, p. 248–266.

[LOW 95] LOWE G., « An Attack on the Needham-Schrœder Public-Key Authentication Protocol », *Information Processing Letters*, vol. 56, n° 3, 1995, p. 131–133.

[LYN 96] LYNCH N., *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, California (USA), 1996.

[MAG 99a] MAGEE J., « Behavioral Analysis of Software Architecture using LTSA », *Proceedings of the 21st International Conference on Software Engineering*, mai 1999.

[MAG 99b] MAGEE J., KRAMER J., *Concurrency : State Models and Java Programs*, John Wiley and sons, 1999.

[MAU 90] MAUW S., VELTINK G., « A Process Specification Formalism », *Fundamenta Informaticae*, vol. XIII, 1990, p. 85–139.

[MAU 93] MAUW S., VELTINK G., Eds., *Algebraic Specification of Communication Protocols*, vol. 36 de *Cambridge Tracts in TCS*, Cambridge University Press, 1993.

[MCM 93] MCMILLAN K., *Symbolic Model Checking*, Kluwer Academic Publ., 1993.

[MIL 80] MILNER R., *A Calculus of Communicating Systems*, vol. 92 de *Lecture Notes in Computer Science*, Springer Verlag, 1980.

[MIL 83] MILNER R., « Calculi for synchrony and asynchrony », *Theoretical Computer Science*, vol. 25, 1983, p. 267–310.

[MIL 85] MILNE G. J., « CIRCAL and the Representation of Communication, Concurrency, and Time », *ACM Transactions on Programming Languages and Systems*, vol. 7, n° 2, 1985, p. 270–298.

[MIL 89] MILNER R., *Communication and Concurrency*, Prentice-Hall, 1989.

[MIL 92] MILNER R., PARROW J., WALKER D., « A calculus of mobile processes », *Journal of Information and Computation*, vol. 100, 1992, p. 1–77.

[NEE 78] NEEDHAM R. M., SCHRŒDER M. D., « Using Encryption for Authentication in Large Networks of Computers », *Communications of the ACM*, vol. 21, n° 12, 1978, p. 993–999.

[NIC 90] NICOLA R. D., VAANDRAGER F. W., « Action versus State based Logics for Transition Systems », GUESSARIAN I., Ed., *Semantics of Systems of Concurrent Processes*, vol. 469 de *Lecture Notes in Computer Science*, p. 407–419, Springer Verlag, avril 1990.

[NIC 94] NICOLLIN X., SIFAKIS J., « The Algebra of Timed Processes ATP : Theory and Application », *Information and Computation*, vol. 114, n° 1, 1994, p. 131–178.

[OLD 91] OLDEROG E. R., *Nets, Terms, and Formulas*, Cambridge University Press, 1991.

[OMG 03] OMG, « OMG Unified Modeling Language Specification – Version 1.5 », rapport n° formal/03-03-01, mars 2003, Object Management Group.

[PAI 87] PAIGE R., TARJAN R. E., « Three Partition Refinement Algorithms », *SIAM Journal of Computing*, vol. 16, n° 6, 1987, p. 973–989.

[PAR 81] PARK D., « Concurrency and Automata on Infinite Sequences », DEUSSEN P., Ed., *Theoretical Computer Science*, vol. 104 de *Lecture Notes in Computer Science*, Springer Verlag, mars 1981, p. 167–183.

[PNU 85] PNUELI A., « In transition from global to modular temporal reasoning about programs », APT K., Ed., *Logics and Models of Concurrent Systems*, Springer Verlag, 1985, p. 123–144.

[PUG 99] PUGH W., « Fixing the Java Memory Model », *Java Grande*, ACM, 1999, p. 89–98.

[PUG 00] PUGH W., « The Java Memory Model is Fatally Flawed », *Concurrency : Practice and Experience*, vol. 12, n° 1, 2000, p. 1–11.

[ROS 97] ROSCOE B., *The Theory and Practice of Concurrency*, Prentice Hall, 1997.

[SAL 70] SALWICKI A., « Formalized algorithmic languages », *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys.,*, vol. 18, 1970, p. 227–232.

[SAN 01] SANGIORGI D., WALKER D., *The pi-calculus : a Theory of Mobile Processes*, Cambridge Universtity Press, 2001.

[SCH 99] SCHNEIDER S. A., *Concurrent and Real-time Systems : the CSP approach*, John Wiley, New York, 1999.

[SIG 00] SIGHIREANU M., « LOTOS NT User's Manual (Version 2.1) », INRIA projet VASY. `ftp ://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, novembre 2000.

[STR 86] STROM R. E., YEMINI S., « Typestate : A programming language concept for enhancing software reliability », *IEEE Transactions on Software Engineering*, vol. 12, n° 1, 1986, p. 157–171.

[STR 90] STROM R. E., BACON D. F., LOWRY A., GOLDBERG A. P., YELLIN D. M., YEMINI S. A., « Hermes : A Language for Distributed Computing », Technical report, octobre 1990, IBM T. J. Watson Research Center, Yorktown Heights, New York.

[TAU 89] TAUBNER D., *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, vol. 369 de *Lecture Notes in Computer Science*, Springer Verlag, 1989.

[VIS 88] VISSERS C., SCOLLO G., SINDEREN M., « Architecture and Specification Style in Formal Descriptions of Distributed Systems », AGGARWAL S., SABNANI K., Eds., *Proceedings of the 8th International Workshop on Protocol Specification, Testing and Verification (Atlantic City, NJ, USA)*, IFIP, North-Holland, 1988, p. 189–204.

[YAN 01] YANG Y., GOPALAKRISHNAN G., LINDSTROM G., « Analyzing the CRF Java Memory Model », *8th Asia-Pacific Software Engineering Conference*, 2001, p. 21–28.