Modularité et symétrie pour les systèmes répartis; application au langage CSP

Luc Bougé

T H È S E présentée

pour l'obtention du

DIPLOME de DOCTEUR d'ÉTAT

à

l'Université Paris 7

Spécialité : Mathématiques

Mention : Informatique

par M^r Luc Bougé

Sujet: Modularité et symétrie pour les systèmes répartis; application au langage *CSP*.

Soutenue le 30 Mars 1987 devant le jury suivant:

Maurice Nivat, président
Krzysztof Apt, rapporteur
Daniel Lehmann, rapporteur
Matthew Hennessy,
Michel Raynal,
Gérard Boudol,
Guy Cousineau.

à dominique, douce
à Jérémie, impatient
à Félix, attentif
à Léa, sourire

Je voudrais remercier ici tous ceux par qui cette thèse a été possible.

Je remercie Maurice Nivat qui m'a accueilli au LITP après ma thèse de troisième cycle, et qui m'a permis d'entreprendre ce travail.

Le materiel de cette thèse trouve son origine dans le travail de Krzysztof Apt. Il a pris le temps de m'initier aux mystères du parallélisme. Il m'a fait découvrir peu à peu les exigences et les joies du métier de chercheur. Je lui dédie donc tout spécialement ce travail, au terme de ces quatre années de travail en équipe.

Daniel Lehmann a accepté de s'intéresser à ce travail. C'est un grand honneur pour moi. Plusieurs résultats de cette thèse découlent de discussions que nous avons eues ensemble.

Matthew Hennessy a bien voulu faire partie de ce jury. J'ai beaucoup d'admiration pour son travail, et j'espère que nous aurons l'occasion de collaborer à l'avenir.

Je suis très heureux que Michel Raynal soit présent dans ce jury aujourd'hui. J'apprécie beaucoup son approche pratique des problèmes réputés théoriques, et son souci pédagogique constant. Son avis sur ce travail m'est très précieux.

J'ai beaucoup appris de Gérard Boudol, et je pense que j'apprendrai encore beaucoup de sa vision de la sémantique du parallélisme. Je suis très heureux qu'il puisse siéger dans ce jury, et donner son appréciation sur mon travail.

Je suis aussi très fier que Guy Cousineau ait accepté de participer à ce jury. Je suis très heureux que des convergences semblent se dessiner entre notre intérêts, et que nous ayions l'occasion de travailler ensemble.

Ce travail doit aussi beaucoup au Département de Mathématiques et d'Informatique de l'Université d'Orléans, et en particulier à Bernard Lorho. J'ai trouvé à Orléans un accueil chaleureux et les moyens matériels de travailler, et plus spécialement de rédiger cette thèse. J'espère que ma présence dans ce laboratoire n'a pas été une gêne trop grande, mais au contraire qu'elle a contribué à lui permettre de prendre l'envergure qu'on lui connaît maintenant. Je souhaite de tout cœur que cette symbiose s'enrichisse encore à l'avenir.

Je remercie tous ceux (et toutes celles!) qui ont contribué de près ou de loin à l'écriture de cette thèse, notamment pour la traduction du texte allemand présenté en annexe. Que tous sachent combien leur aide, mais aussi leur confiance et leurs encouragements m'ont été precieux.

Luc Bougé, 30 Mars 1987

Titre	Title
Modularité et symétrie pour les systèmes répartis; application au langage CSP.	Genericity and symmetry for distributed systems; the case of CSP.

Mots-clé	Key words
systèmes répartis	distributed systems
réseau de processeurs	network of processors
CSP	CSP
modularité	genericity
symétrie	symmetry
diffusion	broadcasting algorithms
élection	leader finding algorithms
terminaison répartie	distributed termination
rendez-vous	handshaking
sémantique partiellement ordonnée	partially ordered semantics

Résumé

L'évaluation des systèmes répartis est habituellement fondée sur des critères numériques relatifs à la quantité d'information échangée au cours des calculs. Nous montrons que ces critères ne sont pas suffisants pour évaluer le degré de répartition des algorithmes répartis usuels. Des critères qualitatifs, spécifiques de la répartition, sont nécessaires.

La modularité exprime que les processeurs du système n'ont initialement aucune connaissance concernant globalement le réseau dans lequel ils sont plongés. La symétrie exprime que des processeurs avec des positions topologiquement équivalentes dans le réseau ont aussi des rôles équivalents dans les calculs.

Nous définissons ces propriétés dans le cadre du langage CSP des processus séquentiels communicants de Hoare. Nous proposons une définition syntaxique pour la modularité. Nous montrons qu'une définition syntaxique de la symétrie n'est pas suffisante. Nous en proposons donc une définition sémantique. Cette définition se réfère implicitement à une sémantique partiellement ordonnée de CSP.

Nous étudions l'existence d'algorithmes de diffusion et d'élection dans les réseaux de processus communicants, qui soient modulaires et symétriques. Nous obtenons de nombreux résultats positifs et négatifs. Ceci conduit en particulier à une évaluation précise du pouvoir expressif de CSP. Nous montrons par exemple qu'il n'existe pas d'implantation des gardes d'émission par des gardes de réception seulement, si la symétrie doit être préservée.

Ces résultats sont enfin utilisés pour proposer une solution modulaire, symétrique et bornée au problème de la détection de la terminaison répartie proposé par Francez.

Abstract

Numerical criteria are usually considered to evaluate distributed systems. They are mainly based on information transfers whithin computations. We show those criteria are not satisfactory in evaluating the degree of distributedness of distributed algorithms. Specifically designed non-numerical criteria are needed.

Genericity expresses that processors have initially no knowledge about the network they are embedded in. Symmetry expresses that processors with topologically equivalent positions in the network play equivalent rôles within computations, too.

We define those properties in the framework of Hoare's language CSP of Commmunicating Sequential Processes. We give a syntactic definition for genericity. We show that such a definition is not appropriate for symmetry. A semantic definition is needed.

We consider then broadcasting and leader-finding algorithms for networks of communicating processes in CSP. We study the existence of generic and symmetric solutions. Many results, positive as well as negative, are carried out. This leads to a precise evaluation of the expressive power of CSP. In particular, we show that output guards cannot be implemented by means of input guards only, as long as symmetry is preserved.

Those results are finally applied to describe a generic, symmetric and bounded solution to Francez' problem of distributed termination detection.

Tables des matières

0. Introduction

Au cours des dernières années, le domaine de l'algorithmique répartie a pris une importance toujours croissante. Les progrès technologiques et les réalités économiques ont ouvert de nouveaux champs d'application encore en pleine évolution.

Les réseaux de transmission de données (cf. [QH 86] pour une introduction détaillée) nécessitent l'introduction de protocoles toujours plus raffinés, que ces réseaux se situent à l'échelle microscopique ("bus" entre processeurs), à l'échelle d'un bâtiment (réseau local) ou à l'echelle d'un pays (TRANSPAC, mais aussi CSNet *etc.*) Les problèmes sont d'ailleurs considérablement accrus par la nécessité d'interconnecter des réseaux hétérogènes, conçus souvent totalement indépendamment.

Dans un autre domaine, il faudrait citer aussi les bases de données réparties. Contrairement au cas des réseaux où le problème est principalement d'acheminer de l'information entre différents sites, il s'agit plutôt dans ce cas de maintenir la cohérence des données au niveau global lors de modifications locales de la base. Les exemples les plus classiques dans ce domaine sont sans doute les systèmes de gestion bancaire ou de réservation aérienne. Dans ces deux cas, l'aspect base de données est étroitement dépendant de l'aspect réseau sous-jacent.

0.1. Evaluation des algorithmes répartis

Devant cette prolifération d'algorithmes répartis, le problème de l'évaluation se pose de manière cruciale. Il existe des dizaines d'algorithmes d'exclusion mutuelle, d'élection ou de détection des interblocages. Mais, paradoxalement, on ne dispose que de très peu de critères permettant de comparer ces algorithmes, ou, plus simplement, d'affiner la spécification du problème pour établir une hierarchie entre eux.

La première difficulté est sans conteste la variété extrême des modèles de calcul. Dans le cas séquentiel, on dispose depuis longtemps d'un modèle de référence, la machine de Turing. La thèse de Church exprime que tout autre modèle "raisonnable" de calcul séquentiel peut être ramené à ce modèle ou à l'une de ses variantes. La situation dans le cas réparti est totalement différente, du moins à l'heure actuelle. Il n'existe pas pour le moment de modèle de référence. Chaque algorithme est développé dans un cadre particulier sans qu'il soit souvent possible de distinguer le principe intrinsèque de l'algorithme des adaptations dues au modèle de calcul sous-jacent. Par exemple, l'algorithmique des systèmes avec mémoire (plus ou moins localement) partagée est très différente de celle des systèmes avec communications point-à-point. La possibilité de créer dynamiquement des processus influe aussi profondément sur la conception des algorithmes. Très peu de choses sont connues sur les relations entre les divers modèles. De même, dans le cas des communications point-à-point, il existe une différence radicale entre les modèles avec communications synchrones et asynchrones.

Une seconde difficulté est que chacun de ces modèles dispose de multiples critères d'évaluation. Ils sont généralement issus de l'algorithmique séquentielle, et ne prennent pas en considération la spécificité de l'algorithmique répartie. Dans le cas des systèmes avec communications point-à-point, on considère habituellement le nombre de messages échangés au cours du calcul en fonction du nombre de processeurs. Ce critère est très insatisfaisant pour plusieurs raisons. Il est clair que, pratiquement, il est très différent d'envoyer un message court ou un message long. Par exemple, certains algorithmes "optimaux" pour cette mesure dans le cadre des problèmes byzantins utilisent des messages de longueur exponentielle (DM 86])! En fait, ce critère caractérise la nombre d'échanges d'information nécessaires à l'algorithme, indépendamment de leur contenu.

Un remède consiste à compter globalement le nombre de bits échangés, indépendamment du nombre de messages. Ce critère caractérise donc la quantité d'information échangée, sans référence à la manière dont celle-ci est échangée.

Dans le cas d'un système évoluant de manière synchrone, on peut aussi compter le nombre de phases exécutées par le système. Ceci constitue une mesure du degré de connaissance nécessaire aux processus pour prendre une décision donnée ([DM 86]).

Tous ces critères peuvent bien sûr être appliqués de diverses manières. Il est possible de considérer la pire des exécutions possibles pour un algorithme donné. Il est aussi possible de considérer la moyenne sur toutes les exécutions possibles selon une certaine loi de probabilité. Très peu de choses sont connues sur les relations entre ces différents critères d'évaluation. Il semble cependant que tous ces critères aient une interprétation naturelle en terme de la diffusion de l'information au sein des systèmes (*cf.* [CM 85a] pour une présentation générale de ce point de vue).

Une troisième difficulté est que ces critères multiples ne rendent pas compte de manière satisfaisante de la spécificité de l'algorithmique répartie, à savoir précisément la répartition des données et du contrôle entre différents processus. Ainsi, pour les critères cités plus haut, un algorithme simulant une gestion centralisée au sein d'un système réparti pourra être considéré comme meilleur qu'un algorithme vraiment réparti! L'exemple suivant permettra de mieux comprendre ce paradoxe.

0.2. Un exemple

Considérons le problème de la multi-diffusion qui sera étudié en détail au chapitre 6. Soit G un réseau bidirectionnel connexe. Ses noeuds seront notés a, b *etc.* A chaque noeud u est associé un processus P_u. Ces processus communiquent par échange de messages le long des côtés de G. Initialement, chaque processus P_u possède une donnée d_u. On veut que tous les processeurs finissent par connaître toutes les donnés (v, d_v), où v parcourt l'ensemble des sommets de G.

Ce problème a été étudié par de nombreux auteurs, et nous décrivons ci-dessous brièvement 4 solutions.

Solution 1

On choisit un sommet de G, disons a. On choisit un arbre recouvrant T de G, de racine a. L'idée est de faire remonter toutes les données jusqu'à la racine, puis de faire redescendre le résultat ainsi accumulé.

Chaque processus P_u attend un message de chacun de ses fils dans T. Il accumule ces messages dans une variable set_u. Il y ajoute sa propre donnée (u, d_u) et envoie le tout à son père dans T. Finalement, P_a possède l'ensemble des données dans sa variable set_a. Il envoie alors cette donnée à chacun de ses fils dans T et termine. Chaque processus P_u attend de recevoir ce message de son père dans T, le recopie dans sa variable set_u, l'envoie à chacun de ses fils et termine.

Solution 2

On choisit un sommet de G, disons a. L'idée est de rammasser toutes les données par un premier parcours de G de type séquentiel en profondeur d'abord ("sequential depth-first traversal"), puis de les diffuser par un second parcours de ce type.

Chaque processeur P_u, sauf P_a, attend donc un message d'un de ses voisins. Il appelle alors celui-ci son père. Il recopie le message reçu dans sa variable set_u et y ajoute sa propre donnée (u, d_u). Pour chacun de ses autres voisins, il fait alors les actions suivantes. Si le nom de ce voisin apparaît dans set_u il ne fait rien. Sinon, il envoie à ce voisin une copie de set_u, attend de ce voisin un message, et ajoute ce message à set_u. Lorsque ceci est fait, il renvoie une copie de set_u à son père. Lorsque P_a a ainsi interrogé tous ses voisins, il possède dans sa variable set_a l'ensemble de toutes les valeurs. Il initialise alors une seconde vague du même type pour diffuser ces valeurs à tous les processus.

Solution 3

On procède comme dans la solution précédente, en utilisant cette fois un parcours non séquentiel. Chaque processeur P_u attend un message d'un de ses voisins qu'il appelle son père. Il copie ce message dans sa variable set_u, y ajoute sa propre donnée (u, d_u), et en envoie une copie à chacun de ses autres voisins. Il attend une réponse de chacun d'eux, les accumule dans set_u, et renvoie le tout à son père. Si un autre proceseur l'interroge, il répond en renvoyant simplement le message qu'il a reçu.

Solution 4

On utilise une stratégie de propagation d'onde. Chaque processus P_u envoie à tous ses voisins le contenu de sa variable set_u, initialement (u, d_u), après chaque incrémentation de celle-ci (y compris celle ayant lieu initialement). Sur réception d'un message, il ajoute le contenu du message à sa variable set_u. On remarquera que la terminaison de cet algorithme pose un problème délicat (*cf.* chapitre 6).

Evaluation

Si l'on considère le nombre de messages échangés, les algorithmes ci-dessus se classent comme suit. Soit N le nombre de sommets de G et E le nombre de côtés. La première solution est $O(N)$. La seconde l'est aussi, mais avec des messages beaucoup plus longs. La troisième est $O(E)$, mais avec des messages courts. Enfin la quatrième est $O(E * N)$. Il semble donc que la première solution soit la meilleure pour cette mesure.

Si maintenant nous regardons ces mêmes algorithmes d'un point de vue qualitatif, le classement est très différent. Dans la première solution, le processeur P_a joue un rôle très particulier. Il centralise en fait l'information et initialise la diffusion. Il apparaît donc comme le moniteur du système, et on peut parler d'une gestion centralisée dans un univers réparti. D'autre part, chaque processeur a besoin pour commencer son travail d'une information concernant la topologie globale du réseau, à savoir sa position dans l'arbre recouvrant. Le comportement d'un processeur ne dépend donc pas uniquement de son environnement local dans le réseau, mais aussi de données globales.

La deuxième solution utilise une gestion centralisée du même type que ci-dessus. De plus, le calcul est conduit de manière séquentielle. A chaque instant, au plus un processus est actif en ce sens que l'arrêt de celui-ci cause l'arrêt du système. Par contre, les processus n'ont besoin d'aucune information globale concernant le réseau pour conduire leurs calculs. Le comportement d'un processus ne dépend que de ses voisins immédiats.

La troisième solution présente les mêmes caractéristiques que la deuxième, mis à part la séquentialité. Les différentes diffusions sont conduites en parallèle, ce qui peut entrainer une certaine redondance du parcours, expliquant ainsi la complexité obtenue plus haut.

Dans la quatrième solution, par contre, tous les processus jouent des rôles équivalents. En particulier, aucun d'eux n'a besoin d'initialiser de manière exclusive l'algorithme. Comme ci-dessus, la diffusion se fait de manière concurrente. Enfin, les processus n'ont pas besoin d'information concernant la tologie globale du réseau. Selon ces critères, la solution 4 semble donc la meilleure!

Cette discussion montre combien les critères habituels quantitatifs peuvent conduire à une évaluation imparfaite, privilégiant de fait les comportements centralisateurs séquentiels et dépendant de contraintes globales. D'autres critères, de type qualitatif cette fois, sont nécessaires pour caractériser, non plus l'efficacité des calculs, mais leur structure. L'objet de cette thèse est précisément l'étude de deux tels critères, la modularité et la symétrie.

0.3. Modularité et symétrie

L'objet de cette thèse est l'étude de certains critères d'évaluation informels présentés ci-dessus. Il apparaît en effet que ces critères sont utilisés implicitement dans de très nombreux travaux pour montrer que tel algorithme est meilleur que tel autre. Cependant, malgré ce rôle crucial, en pratique ils ne semblent pas avoir reçu l'attention qu'ils mériteraient de la part des théoriciens. Nous avons identifié deux critères parmi sans doute beaucoup d'autres : la modularité et la symétrie.

La **modularité** caractérise la quantité d'information nécessaire aux processus pour prendre part aux calculs. Nous dirons qu'un algorithme est modulaire si, initialement, les processus n'ont besoin que d'informations locales concernant la topologie du réseau dans lequel ils sont plongés. Ils ne peuvent alors accéder à aucune information globale telle que le nombre de noeuds de ce réseau, l'existence d'une sous-topologie distinguée (anneau hamiltonien, réseau en étoile *etc.*) ou l'existence d'un noeud de nom donné. Chaque processus peut alors être vu comme une "puce", caractérisée uniquement par un certain nombre de paramètres topologiques locaux. L'algorithme est modulaire si tout assemblage consistant de tels circuits élementaires conduit à un algorithme correct.

Il faut remarquer ici que la notion de localité dépend fortement du formalisme utilisé. Certaines informations pourront être considérées comme locales par certains formalismes alors que d'autres les considèreront comme globales. Ce point sera discuté au chapitre 3. La notion d'information (aussi appelée connaissance) est délicate à manipuler directement dans un cadre algorithmique (on notera cependant les avancées récentes de [DM 86] entre autres). L'une des difficultés est de relier la connaissance possédée par le programme avec sa forme syntaxique. En effet, l'information peut apparaitre explicitement sous forme de valeurs conservées dans des variables. Mais elle peut aussi être implicitement contenue dans les structures de contrôle des processus. Les techniques de transformation de programmes séquentiels montrent d'ailleurs qu'il existe des relations étroites entre ces deux modes. Pour éviter d'attaquer directement ce problème complexe,

nous préférons décrire la modularité comme une propriété différentielle. Plutôt que de décrire la modularité comme l'absence d'information globale, nous considérerons, la faculté d'adaptation des processus à d'autres environnement globaux qui invalideraient ces informations. De ce point de vue, la modularité exprime non pas l'absence d'informations globales, mais le fait que les processus n'utilisent pas de telles informations, ce qui, du point de vue de l'observateur externe, revient au même. Pour résumer cette discussion, il nous semble que la modularité est une propriété importante, puisqu'elle exprime le degré d'indépendance des processus par rapport à leur environnement.

La modularité est donc une propriété essentielle de tout système destiné à évoluer, en particulier lors de reconfigurations à la suite de pannes. Parmi les solutions présentées ci-dessus, la première n'est pas modulaire puisque les processus doivent connaître leurs fils dans l'arbre recouvrant T. Par contre, les solutions 2, 3 et 4 sont modulaires. Les processus sont déterminés par leur nom et ceux de leurs voisins. La modularité est un concept statique caractérisant la quantité d'information nécessaire initialement aux processus. La symétrie est un concept dynamique. Elle caractérise la manière dont l'information se propage entre les processus au cours des calculs. Prenons une analogie. En mécanique des gaz, la notion d'isotropie joue un rôle fondamental. Un milieu est dit **isotrope** s'il n'y a pas de direction privilégiée en son sein. Si, à un certain moment, on observe une molécule se déplaçant vers la droite, il aurait tout aussi bien pu se produire qu'elle se déplaçât vers la gauche. Notons que ceci ne signifie pas qu'à chaque instant elle peut se déplacer vers la droite ou vers la gauche. Mais globalement, au niveau de tous les comportements possibles, il n'y a pas de direction privilégiée.

La **symétrie** exprime une idée analogue. Un algorithme présente un caractère symétrique s'il n'y a pas de processus statiquement privilégié au cours des calculs. Si un processus P_u acquiert un certain privilège au cours d'un calcul C (par exemple, P_u gagne l'élection) alors, pour tout autre processus P_v, il existe un certain calcul C' tel que, au cours de C', P_v acquiert ce privilège. En particulier, ceci interdit tout privilège fondé sur un ordonnancement total des noms de processus. La symétrie n'interdit pas aux processus de connaitre leurs noms, mais elle leur interdit de s'en servir pour établir une priorité entre eux. La symétrie exprime que tout privilège doit être dynamiquement négocié, les processus ayant

les mêmes droits et les mêmes devoirs

au cours de cette négociation. Cependant, nous introduisons un correctif à ces exigences. En effet, certains processus, du fait de leur position privilégiée dans le réseau, sont naturellement amenés à présenter un comportement privilégié (le processus placé au centre d'un réseau en étoile par exemple). L'exigence formulée plus haut ne sera donc appliquée qu'aux processus P_v ayant dans le réseau une position équivalente à celle de P_u. La symétrie exprime donc que les processus ayant des positions équivalentes dans le réseau présentent aussi des comportements équivalents. Il s'agit donc d'une adéquation entre les caractéristiques statiques d'un système et ses caractéristiques dynamiques.

La symétrie nous semble une propriété fondamentale des systèmes répartis. Mis à part l'aspect "esthétique" de la symétrie (mis en avant dans [LR 81] et [CLP 83] pour la symétrie syntaxique), il semble clair que des problèmes ayant une spécification symétrique (*cf.* la multi-diffusion ci-dessus) devraient avoir des solutions symétriques. De fait, au delà d'un certain nombre de processus, seules ces solutions semblent pratiquement réalisables (algorithmes systoliques). Pour ce type d'algorithme, la symétrie permet de produire des preuves économiques et fiables. L'absence de symétrie conduit par contre à une multiplication des cas nécessitant une étude séparée. Ce phénomène est d'ailleurs bien connu en mathématiques (preuves par symétrie). La symétrie joue aussi un rôle crucial dans la résistance aux pannes. En effet, si un processeur privilégié tombe en panne, un autre processeur doit être capable d'assumer ce privilège. De fait, tous les algorithmes de type "byzantin" présentent un haut degré de symétrie. Les solutions intrinsèquement optimales de [MT 86] sont même parfaitement symétriques. Enfin, un dernier argument en faveur de la symétrie est d'ordre théorique. La symétrie se révèle être un outil extrêmement précis de classification. L'étude des implantations symétriques d'un langage dans un autre est un moyen très efficace d'évaluer le pouvoir expressif des langages répartis. Ce point sera amplement démontré au chapitre 7.

0.4. Le Modèle

Comme nous l'avons vu plus haut, il n'existe pas (encore ?) de modèle "universel" pour le calcul réparti. Dès lors, le choix d'un modèle est une étape nécessaire et déterminante dans tout travail sur ce sujet.

Nous avons choisi de conduire notre étude dans le cadre du langage CSP des processus séquentiels communicants de Tony Hoare ([Hoare 78]). Une description du langage et de sa sémantique opérationelle sont présentées au chapitre 1. Le choix de CSP est le résultat de plusieurs motivations.

Motivations historiques tout d' abord. Le point de départ de ce travail est l'étude du problème de la détection de la terminaison répartie. En effet, de très nombreux algorithmes ont été proposés sans que l'on ait de critère permettant de les évaluer ou de les comparer. Cette situation est d'autant plus étrange que ces algorithmes sont en fait issus d'un petit nombre de souches, chaque version étant obtenue par modification (optimisation) de la version précédente. Notre but était donc de fournir des critères formels permettant de mettre un peu d'ordre dans ce foisonnement d'algorithmes. Le problème ayant initialement été formulé dans le cadre de CSP ([Francez 80]), ce cadre a été conservé.

Nos critères ont donc aussi tout naturellement été développés dans ce cadre. D'autre part, cette motivation nous a conduit à nous abstenir de tout aménagement de CSP, qui aurait parfois simplifié notre travail. Nous avons tenu à travailler avec la version du langage effectivement utilisée en pratique. De ce point de vue, notre travail peut être qualifié de recherche appliquée.

Mais nous avons vite constaté que les concepts de modularité et de symétrie peuvent trouver des applications théoriques importantes. La principale est sans doute l'une des constructions les plus discutées de CSP, à savoir la notion de garde d'émission. Hoare avait initialement rejeté cette construction considérant qu'elle n'est pas efficacement implantable (le même choix a été fait pour Occam [Inmos 84]) . Cependant, l'usage des gardes d'emission s'est révélé très utile, et des protocoles ont été proposés permettant d'implanter les gardes d'émission. Le problème se posait donc d'évaluer avec précision le pouvoir expressif de cette construction. Notre travail nous permet de répondre à cette question, en montrant qu'il n'existe pas d'implantation symétrique des gardes d'émission. En ce sens, notre travail peut être qualifié de recherche théorique.

Enfin, le problème se posait de trouver une solution satisfaisant nos critères de modularité et de symétrie pour le problème de la détection de la terminaison répartie en CSP. Le chapitre 8 est consacré à la description d'une telle solution.

Ces choix conditionnent crucialement tout notre travail. Il nous semble que les points principaux soient les suivants. Au niveau de la définition de la modularité, nous utilisons la notion d'environnement local d'un processus. Du fait du rôle des identificateurs de processus en CSP, nous devons considérer que cet environnement comprend le nom du processus et les noms de ses voisins entrants et sortants, ce qui semble exagéré. Dans de nombreux formalismes, un processeur envoie ses messages non pas à ses voisins mais à un canal auquel il accède par un nom local. La connexion entre canaux entrants et sortants est assurée par une relation globale au réseau, indépendante des processeurs (*cf.* [KKM 86] par exemple) . Ceci est souvent exprimé en posant que les processeurs ne peuvent distinguer entre les différents canaux adjacents. En ce cas, la notion d'environnement local se réduit au nom du processus et aux noms des canaux adjacents. Plus généralement la notion d'identificateur de processus joue un rôle déterminant en CSP, souvent considéré comme exagéré (la version abstraite de CSP, TCSP corrige ce défaut, *cf.* [HBR 84]) . Cette particularité se reflète dans tout notre travail.

La notion de symétrie que nous proposons est elle aussi influencée par le modèle particulier dans lequel nous travaillons. Les systèmes répartis en CSP que nous considérons admettent de manière naturelle un graphe de communication, qui peut être vu comme le réseau sous-jacent au système. Les particularités syntaxiques de CSP (tableaux de processus notamment) font qu'il ne semble pas possible de donner une définition raisonnable de la symétrie au niveau syntaxique. Nous sommes donc conduit à nous placer au niveau sémantique. La symétrie syntaxique est alors vue comme une condition suffisante de symétrie. Cette condition est cependant très restrictive en général, même si elle semble naturelle dans la pratique. Elle interdit, de fait, d'utiliser les identificateurs de processus dans des prédicats autres que l'égalité, même de manière cachée.

Tout au long de ce travail, nous avons essayé de distinguer nettement ce qui est dû aux contraintes techniques de CSP, de l'intuition générale qui nous guide. Nous pensons que des notions de modularité et de symétrie peuvent être définies de manière anologue dans de très nombreux formalismes répartis, asynchrones

notamment.

0.5. Les Résultats

Nous considérons essentiellement deux types d'applications de modularité et de symétrie:

1) pour la spécification des algorithmes souhaités ou l'évaluation d'algorithmes préexistants;
2) pour l'étude des limites du pouvoir expressif du langage que nous utilisons, en montrant que certaines spécifications ne sont pas réalisables.

Cette dualité entre résultats positifs et négatifs se retrouvera tout au long de ce travail.

Ces notions vont aussi nous permettre d'affiner notre perception des paradigmes qui fondent la conception des algorithmes répartis. En particulier, nous nous attacherons à mettre en évidence les compromis entre différentes notions telles que le degré de parallélisme, le degré de non déterminisme, la symétrie, la modularité, la richesse du langage (en particulier présence de gardes de communication), le degré de connexité du réseau sous-jacent *etc.*

Les différents problèmes que nous étudions permettent d'établir de manière assez précise comment ces diverses notions interagissent. L'un des meilleurs moyens pour étudier une notion abstraite est de résoudre un problème où elle intervient de manière cruciale, un problème qui semble capturer l'essence de cette notion. Ainsi a-t-on par exemple procédé pour les notions de non déterminisme et de répartition, en définissant le problème de l'exclusion mutuelle (*cf.* [Dijkstra 72]) . De même, le parallélisme a été mieux compris au travers du célèbre problème des "dining philosophers" proposé par Dijkstra et plus généralement de l'équité (*cf.* [AO 84] pour une approche récente). Nous entendons procéder de même pour les notions abstraites de symétrie et de modularité. Une difficulté supplémentaire est l'interaction fréquente de ces deux notions.

La notion de modularité sera abordée au travers du problème de la **diffusion** . Il s'agit de permettre à un processeur donné, l'initiateur, de transmettre une valeur à tous les autres processeurs du réseau. La modularité correspond ici à la restriction suivante. L'initiateur ne peut utiliser d'information préalable sur le réseau. Il doit pouvoir s'adapter à n'importe quel réseau dans lequel il serait plongé. De plus, nous exigerons autant que possible que tous les processeurs autres que l'initiateur aient des rôles similaires dans cette diffusion. En fait, un tel algorithme de diffusion conduit naturellement à une **multi-diffusion** , où chaque processeur diffuse, en même temps, sa valeur à tous les autres. La multi-diffusion n'est autre qu'une symétrisation d'une diffusion simple. Lorsque la valeur diffusée par un processeur est précisement son environnement local dans le réseau, on obtient un algorithme d'apprentissage. Par un tel algorithme, chacun des processeurs du réseau apprend la topologie globale de celui-ci à partir de connaissances purement locales. On a donc transformé la possession d'informations préalables globales en acquisation dynamique et coopérative de ces mêmes informations. On peut qualifier cette transformation de **modularisation** .

La possibilité de cette modularisation est étroitememt liée au degré de connexité du réseau qui doit permettre l'acheminement d'une masse suffisante d'information entre les processeurs. De plus si l'on exige la symétrie, les conflits nés de la coopération doivent pouvoir être résolus dynamiquement, ce qui est lié au pouvoir expressif du langage. Enfin, un point extrêmement délicat est la terminaison de cet apprentissage. En effet, s'il est facile de permettre à chaque processeur d'apprendre le réseau entier, il est autrement plus difficile de lui permettre de déterminer, à partir de paramètres locaux seulement, que son rôle dans cet apprentissage collectif est terminé. Il s'agit en fait d'une instance très représentative du problème de la **terminaison répartie**, où la détection de la terminaison coûte "plus cher" que l'algorithme lui-même! L'existence, dans ce cas précis, d'une condition locale de terminaison dépend étroitement du pouvoir expressif du langage *CSP*.

Le problème de la symétrie sera abordé au travers du problème de l' **élection** . Il s'agit de permettre à un ensemble de processeurs répartis sur un réseau de se mettre d'accord sur le nom de l'un d'entre eux. La symétrie traduit le fait qu'il ne peuvent s'aider d'aucun ordre de priorité défini statiquement pour guider cette évolution. Tous les processeurs occupant des positions équivalentes dans le réseau ont des "poids" équivalents. De plus, nous demanderons, autant que possible, que les algorithmes d'élection décrits soient modulaires, c'est à dire independants du réseau sous-jacent. En fait, il devrait être clair que ces diverses exigences sont contradictoires. La symétrie s'oppose en effet à la notion d'élection qui privilégie le processeur élu . Cet antagonisme peut être résolu au prix de l'exploitation précise des caractéristiques du réseau, ce qui contredit la modularité. Ces antagonismes expliquent les résultats extrêmement sélectifs que nous obtenons.

L'existence d'algorithmes symétriques d'élection est étroitement liée aux caractéristiques topologiques du réseau, et au pouvoir expressif du langage employé. De fait, la présence de gardes d'émission et de réception permet d'encapsuler la nécessaire dissymétrie dans une construction symétrique. Sans cet "artifice" nous montrerons qu'il ne peut, en règle générale, exister de tels algorithmes. Ces résultats nous permettront d'évaluer rigoureusement le pouvoir expressif des gardes de communication en CSP. En résumé, les gardes d'émission et de réception sont sémantiquement primitives, à la différence par exemple de la convention de terminaison distribuée pour les boucles (*cf.* [AF 84]). De plus, la conjugaison des deux types de gardes permet d'encapsuler une certaine forme d'équité, qui reste cependant strictement plus faible que l'ω-équité (*cf.* [Best 84]; on trouvera une introduction générale dans [LPS 81]).

De même que l'étude de la diffusion conduit à des procédures de modularisation, celle de l'élection conduit à des procédures de **symétrisation** . En effet, l'une des principales causes de dissymétrie est l'existence dans les algorithmes d'un processeur "maître" chargé d'initialiser telle ou telle action. Que ce processeur soit élu de manière symétrique et l'algorithme redevient symétrique. On a donc remplacé un ordre de priorité statique entre les processeurs par un ordre défini dynamiquement de manière répartie.

0.6. Plan du travail

Le chapitre 1 a été conçu comme une introduction générale au langage CSP. Il peut être omis par les lecteurs déjà familiers de ce langage. Nous présentons au chapitre 2 les outils théoriques nécessaires dans ce travail: notion de graphe de communication, notion de calcul, restriction, fusion et commutation de calculs, notion d'équivalence causale.

La notion de modularité est présentée au chapitre 3, et celle de symétrie au chapitre 4.

Le chapitre 5 présente les notions de composition séquentielle répartie et d'itération répartie. Nous montrons que ces notions préservent la modularité et la symétrie. Ces notions sont utilisées pour décrire des procédures de modularisation et de symétrisation.

Nous étudions au chapitre 6 le problème de la diffusion modulaire, et au chapitre 7, celui de l'élection symétrique. Des résultats d'existence et de non existence sont obtenus en fonction de la topologie du reseau, et de l'utilisation des gardes de communication.

Le chapitre 8 est consacré à la description d'une solution modulaire et symétrique au problème de la détection de la terminaison répartie. Elle est fondée sur une généralisation de l'algorithme des "snapshots" de Chandy et Lamport.

On trouvera en annexe 1 la traduction d'un conte introductif au problème de la terminaison répartie écrit par Friedmann Mattern, de l'Université de Kaiserslautern, R.F.A.

1. Le langage *CSP*

Les notions de modularité et de symétrie qui forment le coeur de ce travail seront étudiées dans le cadre du langage *CSP* ("Communicating Sequential Processes") proposé par Tony Hoare en 1978. Nous avons longuement précisé les raisons de ce choix méthodologique et ses conséquences dans l'introduction de ce travail. Ce chapitre est consacré à la présentation détaillée du langage.

Nous présentons tout d'abord les principes qui ont sous-tendu la conception du langage. Nous nous inspirerons en cela essentiellement de [Hoare 78].

Nous proposons ensuite une introduction pratique au langage à l'aide d'exemples.

Nous donnons enfin une sémantique opérationnelle de *CSP*. Cette sémantique est essentiellement celle décrite par Plotkin ([Plotkin 82]) en termes de systèmes de transitions, à quelques simplifications près. En particulier, nous ne considèrerons pas dans ce travail les problèmes sémantiques liés au parallélisme imbriqué: nous n'utiliserons que des systèmes "plats" formés par la composition parallèle de processus séquentiels:

$$P = [\ P_a \parallel P_b \parallel \ldots]$$

Comme Plotkin, nous ne traiterons pas la convention de terminaison répartie. Ceci ne modifie pas le pouvoir expressif du langage, comme l'ont montré Apt et Francez ([AF 84]). La sémantique opérationnelle proposée par Plotkin est de type entrelacé ("interleaving semantics"). Le parallélisme est représenté par l'entrelacement arbitraire des transitions effectuées par chacun des processus concernés. Les communications sont représentées par l'introduction de multi-transitions effectuées simultanément par deux processus.

1.1. Les 10 principes de *CSP*

Le langage *CSP* a été proposé par Hoare en 1978 pour décrire l'interaction d'une machine avec son environnement, constitué le plus souvent lui-même d'autres machines similaires. les concepts fondamentaux utilisés par Hoare pour rendre compte de ces interactions sont ceux de **communication** et de **synchronisation.** Ces concepts avaient déjà été mis en oeuvre par de nombreux auteurs dès cette époque. Les communications entre machines étaient habituellement modélisées par le partage d'une mémoire commune. Il faut noter que cette approche était assez représentative de l'architecture des systèmes disponibles couramment à cette époque. Un grand nombre de mécanismes de synchronisation avait été proposé, améliorant et raffinant les propositions initiales de Dijkstra, centrées autour de la notion de *sémaphore* ([Dijkstra 72]). Dans ce cadre, la proposition de Hoare apparaît comme originale, puisqu'elle prend le contre-pied de la notion de mémoire partagée en introduisant la notion de processus autonomes et distants, communiquant par échanges explicites de messages. Le langage *CSP* des Processus Séquentiels Communicants peut être caractérisé par les 10 principes de conceptions ci-dessous*.

Principe 1: Non déterminisme

CSP est fondé sur le *langage des commandes gardées* de Dijkstra ([Dijkstra 75]). La notion de base du langage est celle de **processus.** Un processus séquentiel (sans parallélisme interne) peut être assimilé à une commande au sens classique du terme. Les principales structures de contrôle premettant de composer les processus sont l'alternative et l'itération gardées non déterministes, et la mise en parallèle. En particulier, le non déterminisme (local) est introduit au niveau des structures de contrôle, plutôt qu'à celui des expressions comme dans le cas de l'affectation non déterministe "$x :=?$" de Apt et Olderog ([AO 84]). Une composition parallèle termine (proprement) si et seulement si chacun des processus composants termine.

---

* Je remercie Michel Raynal pour les intertitres.

Principe 2: Communications par messages avec processus statiquement nommés

A la différence du langage de Dijkstra, deux processus CSP en parallèle ne peuvent partager de variable. Les interactions entre processus concurrents ne peuvent se produire que lors de communications explicites. Les processus sont statiquement *définis* et *nommés*. Il n'y a pas de possibilité de création explicite de processus.

Principe 3: Deux primitives, envoyer et recevoir

Les deux seules primitives de communication entre processus sont l' **émission** et la **réception** de messages. On considère que le problème de l'acheminement des messages est externe par rapport aux processus eux-mêmes, et relève d'un certain "médium" de communication.

Principe 4: Rendez-vous point à point

L'échange de messages se fait de processus à processus (communication *point-à-point*). Les partenaires sont statiquement définis. le processus émetteur doit nommer statiquement le processsus récepteur, et réciproquement. Notons en passant que cette convention n'a pas été retenue par la plupart des langages de programmation répartie inspirés de CSP: Occam (Inmos 84] par exemple, et dans une certaine mesure Ada ([LeVerrand 82]). Ce point est cependant crucial pour la définition d'une notion raisonnable de projection d'un calcul sur un processus. L'échange des messages est **synchrone.** Il ne peut se produire que lorsque les deux partenaires sont prêts à le réaliser. Chaque partenaire, l'émetteur comme le récepteur, peut donc être amené subir un certain retard dans son exécution, et ce retard peut même être infini si la communication souhaitée n'est jamais possible. Lorsque l'échange a lieu, il est *atomique* (non sécable) pour les partenaires. L'émission et la réception débutent et se terminent *en même temps*. Au moins en pratique, les processus non concernés par cette communication peuvent progresser pendant la communication. Au niveau théorique, on considère que la communication est *instantanée* . On peut montrer que ces deux conceptions conduisent à des observations équivalentes (*cf.* [BK 85]). Ce type de communication s'oppose au mode **asynchrone** où l'émission peut avoir lieu indépendamment de la réception, et ne peut donc être bloquante.

Principe 5: Gardes de communication

En plus des gardes booléennes introduites par Dijkstra dans son langage, CSP admet des **gardes de communication.** En plus d'une éventuelle expression booléenne, ces gardes peuvent contenir une commande de communication. Une telle garde ne peut être sélectionnée ("passée") que si l'expression booléenne est satisfaite et si la communication peut se réaliser dans l'état courant. Par le principe 2, la première condition est **stable** . Une expression booléenne satisfaite dans un état du système reste satisfaite tant que le processus la contenant n'a pas modifié son état *local* .

Dans la version originale de CSP, seules les commandes de réception peuvent apparaître dans les gardes de communication. Cette restriction garantit la stabilité de la seconde condition ci-dessus. En effet, si une communication est possible dans l'état courant, elle continue alors à être possible tant que le processus récepteur n'a pas modifié son état. Le processus récepteur maîtrise donc la communication, et le choix d'une communication (réception) parmi plusieurs communications possibles peut être réalisé localement par ce processus. Celui-ci sait en effet que tant qu'il n'aura pas pris de décision, les conditions déterminant celle-ci restent inchangées. Cette restriction autorise donc des implantations simples et efficaces du langage. C'est pourquoi elle a été reprise par les successeurs de CSP destinés à être pratiquement utilisés, Occam ou Ada par exemple.

Cependant, comme Bernstein le suggère ([Bernstein 80]), il est souvent agréable (et même parfois nécessaire, comme nous le verrons bientôt!) de pouvoir disposer de commandes de réception et d'émission dans les gardes. La plupart des auteurs ont suivi cette opinion, qui ne pose aucun problème sémantique particulier au niveau où nous travaillons. A la suite de Plotkin, nous considèrerons cette extension. Nous distinguerons donc trois "dialectes" de CSP:

- CSP_{no} où aucune garde de communication n'est admise;
- CSP_{in} où seules les gardes de réception sont admises;
- $CSP_{i/o}$ où gardes d'émission et gardes de réception sont admises.

Principe 6: Convention de terminaison répartie

La **convention de terminaison répartie** a été introduite par Hoare pour faciliter la programmation en *CSP*. Elle spécifie qu'une garde de communication adressant un processus déjà terminé doit être considérée comme fermée, à l'instar d'une garde booléenne s'évaluant à faux. Cette convention permet une programmation très élégante de systèmes mettant en présence des processus "serveurs" et des processus "clients". La terminaison des clients entraine alors naturellement la terminaison des serveurs, sans que le programmeur ait explicitement à prévoir celle-ci.

Cette convention est certes très utile dans la pratique, et a été au moins considérée dans la conception des langages Occam et Ada. Au niveau théorique, elle se révèle cependant extrêmement lourde à manipuler à cause de son caractère non local et non causal. De plus, comme l'ont montré Apt et Francez ([AF 84]), cette convention n'augmente pas le pouvoir expressif de *CSP*. Conformément au choix fait par Plotkin, nous ne la considèrerons donc pas dans ce travail.

Principe 7: Canaux de communication

La détermination des communications possibles est réalisée par un mécanisme de filtrage statique. Pour simplifier notre propos, nous considèrerons qu'à chaque commande de communication est associé explicitement un nom de canal, défini statiquement. Ce nom peut être vu comme le type des données échangées par cette commande. Deux processus peuvent donc être liés par un certain nombre de canaux véhiculant des informations différentes. Un processus peut exiger sélectivement une communication sur l'un de ces canaux. Les noms de canaux ou types étant définis statiquement, on peut déterminer tous les couples de commandes de communication capables de se synchroniser sur un échange. Remarquons que cette détermination ne produit qu'un sur-ensemble des communications pouvant réellement se produire. Ce sur-ensemble est d'autant plus précis que l'analyse statique est plus fine (*cf.* [Apt 83]). Une détermination parfaitement précise est bien sûr non calculable.

Une autre manière de voir ces types est de considérer qu'ils matérialisent les préconditions liées à la communication. Une communication ne peut se produire que si ses préconditions sont vérifiées par les deux partenaires. Plutôt que d'admettre que la réception d'un message inopportun puisse produire une erreur, on préfère considérer plus simplement qu'un tel message ne peut être échangé. En d'autres termes, un message ne peut être émis que s'il est garanti que sa réception ne provoquera pas d'erreur.

Principe 8: Absence de temps et d'état global

Conformément au concept-même de système réparti, il est admis que les processus évoluant en parallèle n'ont pas conscience d'un temps global au système. Seule la séquentialité des processus détermine un temps local à chacun d'eux. Ces différents "temps" ne sont reliés entre eux que par les relations de synchronisation induites par les communications. En particulier, un processus en attente d'émission ou de réception ne peut avoir conscience du "temps" écoulé lors de cette attente. Il ne peut différencier une attente nulle (le partenaire était prêt) d'une attente très longue. Il ne peut donc tester si une communication est possible dans l'état courant autrement qu'en réalisant effectivement cette communication.

A proprement parler, cette notion de temps doit être remplacée par celle de **causalité** , introduite par Lamport ([Lamport 78]). Deux événements sont causalement ordonnés si l'on peut déduire des relations de séquentialité et de synchronisation que l'un s'est produit *nécessairement* avant l'autre. Pour tout observateur externe du système global, ils apparaîtront alors dans cet ordre, le premier étant alors considéré comme la "cause" du second. Par contre, l'ordre d'occurrence de deux événements non ordonnés causalement peut varier d'un observateur à l'autre. En particulier, les processus eux-mêmes ne peuvent être conscients de cet ordre. L'objet de la notion de "snapshot" développée au chapitre 8 est précisément de permettre aux processus de prendre conscience de l'un des ordonnancement possibles, sans qu'ils puissent toutefois déterminer si cet ordonnancement a réellement été observé par un observateur donné.

La convention de terminaison répartie est un exemple typique de notion non causale en ce sens que la terminaison d'un processus est un événement local à ce processus. Un processus ne peut donc en aucun cas déduire des relations de synchronisation qu'un autre processus *est* terminé. Il peut tout au plus déduire quue ce processus *va* terminer. Cette remarque confirme notre décision de ne pas considérer cette convention dans ce travail.

Principe 9: Absence d'équité

Bien que les processus n'aient pas d'échelle de temps commune, il est d'usage de différencier les attentes finies et infinies. Ceci revient à introduire les notions de justice et d'équité qui garantissent que les attentes infinies correspondent à des situations intrinsèquement bloquées, et non à un étirement infini du temps. Incidemment, Hoare souligne que seuls les calculs justes (au sens de "process justice" de [BK 85]) doivent être considérés comme significatifs pratiquement. Cependant, au niveau de la spécification du langage, il n'y a aucune raison de rejeter les calculs injustes.

Principe 10: Graphe de communication statique

Dans la mesure où les processus destinateurs et destinataires sont statiquement definis par les commandes de communication, un système réparti en CSP induit, de manière statique, un graphe de communication. Seuls les processus attachés à deux sommets voisins dans ce graphe peuvent échanger (éventuellement) un message. Ce graphe peut être vu comme le réseau de communication sur lequel est bâti le système. Cette caractéristique s'oppose donc à toute notion de création dynamique de processus au cours du calcul, comme cela est possible en CCS, Ada ou Occam. Ceci n'est envisageable que si les canaux de communication jouissent d'une existence propre, indépendamment des processus qui doivent les déclarer explicitement pour les utiliser. Il nous semble que ce point différencie radicalement CSP de son contemporain CCS conçu par Milner. Cette caractéristique donne à CSP une coloration pratique et opérationnelle, alors que CCS est plutôt considéré comme un outils théorique et algébrique de spécification. Le langage TCSP ("Theoretical CSP"), proposé par Hoare, Brookes et Roscoe ([HBR 84], [HO 85]) peut être vu comme un essai de synthèse, tout comme la langage Occam ([Inmos 84], [Roscoe 84]). Dans les deux cas, la notion implicite de canal retenue par CSP a été remplacée par une notion explicite, conférant ainsi aux canaux une existence autonome.

1.2. CSP par l'exemple

Nous proposons ici une introduction informelle au langage CSP. Une sémantique opérationnelle précise du langage sera décrite en section 1.3. Nous nous sommes ici attaché aux aspects du langage utilisés dans ce travail. D'autres aspects importants ont été laissés dans l'ombre. C'est en particulier le cas pour les problèmes liés à l'équité des calculs infinis, dont l'étude précise dépasse le cadre de ce travail.

1.2.1. Les commandes

Un *réseau* de communication G est un graphe (fini) orienté avec au plus un arc d'un sommet à l'autre. Les sommets de G sont appelés *processeurs*. Les arcs de G sont appelés *canaux* unidirectionnels de communication. On identifie deux canaux unidirectionnels de direction opposée avec un canal bidirectionnel.

Un *système réparti* P construit sur G est obtenu en attachant à chaque processeur u de G un processus noté P_u. On confondra géneralement le processeur du réseau et le processus du système qui y est attaché.

Un *processus* est un programme séquentiel. Chaque processus gère une mémoire privée à laquelle les autres processus ne peuvent accéder, ni en écriture, ni en lecture. Tous les processus d'un système s'exécutent en parallèle, à leur propre vitesse, sans autre restriction de synchronisation que celles spécifiées ci-dessous.

Un processus est construit à partir des commandes d'affectation usuelles, disons du langage Pascal pour fixer les idées. L'instruction vide est traditionnellement notée *skip*. En plus de ces commandes, deux commandes d'affectation à distance ("remote assignment") sont disponibles.

émission

L'exécution de la commande

$$P_v!chan(val)$$

par P_u provoque l'envoi du mesage $chan(val)$ de P_u vers P_v, en supposant que le réseau G ait un arc de u vers v. *chan* peut être vu comme un canal (virtuel) entre P_u et P_v ou comme le type du message. *val* est la valeur émise.

réception

L'exécution de

$$P_u?chan(x)$$

par P_v provoque la réception d'un message $chan(val)$ en provenance de P_u, en supposant que G ait un arc de u vers v. La valeur val est affectée à la variable x de P_v. L'effet net est donc une affectation à distance d'un valeur de P_u à une variable de P_v.

Les communications en CSP sont *synchrones* en ce sens que l'émission et la réception d'un même message sont simultanées. P_u ne peut émettre un message $chan(val)$ vers P_v si celui-ci n'est pas prêt à recevoir un message $chan(x)$. Réciproquement, P_v ne peut recevoir un message si P_u n'est pas prêt à émettre un message correspondant. Un processus prêt à communiquer sans que son partenaire (ou ses partenaires, comme nous le verrons) le soit est simplement mis en attente. Cette attente peut éventuellement être infinie, auquel cas on parle de blocage ("lockout"). Il est important de remarquer qu'un processus ne peut "avoir conscience" de la durée de son attente. En effet, les vitesses des différents processus sont totalement arbitraires. Un message vide $chan()$ est appelé *signal*. Dans le cas de l'échange d'un signal, émetteur et récepteur jouent des rôles équivalents, chacun pouvant être retardé par l'autre.

Considérons le réseau G_1 ci-dessous.

$$a \longleftrightarrow b$$

Considérons les systèmes suivants construits sur G_1.

P_a :: [x:= 2 ; $P_b!chan(x)$; $P_b?chan(x)$]
P_b :: [y:= 3 ; $P_a?chan(y)$; $P_a!chan(y+1)$]

Ce système termine avec $x = 3$ et $y = 2$. remarquer que la valeur initiale de y dans P_b est indifférente.

Considérons les systèmes suivants.

P_a :: $[P_b!chan(2)$; $P_b?chan(x)]$
P_b :: $[P_a!chan(3)$; $P_a?chan(y)]$

Ce système se bloque en attente infinie du fait des communications synchrones. Ce ne serait pas le cas si les communications étaient asynchrones (émission non bloquante).

P_a :: $[P_b!chan_1()]$
P_b :: $[P_a?chan_2()]$

Ce système se bloque aussi puisque les processus ne sont pas en attente sur le *même* canal.

1.2.2. Structures de contrôle

En plus des sructures de contrôle séquentielles habituelles, CSP dispose de deux constructions non déterministes inspirées du langage des commandes gardées de Dijkstra. Ces constructions sont fondées sur la notion de *commande gardée*. Une commande gardée est constituée d'une commande prefixée par une garde

guard -> *command*

Une garde *guard* est constituée d'une suite (conjonction) d'expressions booléennes (la constante *true* par défaut), suivie éventuellement d'une commande de communication. On parle alors d'une garde de communication (réception ou émission suivant le cas), et d'une garde (purement) booléenne sinon.

Une garde est *ouverte* si ses expressions booléennes sont satisfaites dans l'état courant, et si son éventuelle commande de communication peut être exécutée dans l'état courant. Une garde est *fermée* si l'une au moins de ses expressions booléennes n'est pas satisfaite. Il est donc possible qu'une garde de communication ne soit ni ouverte ni fermée. On dit alors qu'elle est *entr'ouverte*.

Nous distingons plusieurs dialectes de CSP, qui se caractérisent par le type de garde disponible. En CSP_{no}, seules les gardes purement booléennes sont autorisées. En CSP_{in}, on autorise les gardes booléennes et les gardes de réception, mais pas les gardes d'émission (c'était le choix fait originellement par Hoare, et c'est aussi le choix du langage Occam). Enfin, en $CSP_{i/o}$, on autorise les gardes booléennes et les gardes d'émission et de réception.

Construction de sélection

L'alternative non déterministe est de la forme

```
[ guard_1 -> command_1
▮ ...
▮ guard_p -> command_p]
```

L'une des gardes ouvertes est choisie et l'éventuelle communication est exécutée. Le contrôle est ensuite passé à la commande associée. Si toutes les gardes sont fermées, une erreur se produit et le calcul avorte. Si aucune garde n'est ouverte, mais toutes ne sont pas fermées, le proceesus est simplement mis en attente. Remarquer que, dans ce dernier cas, il peut y avoir plusieurs partenaires potentiels.

Construction d'itération

L'itération non déterministe est notée

```
*[ guard_1 -> command_1
▮  ...
▮  guard_p -> command_p]
```

La sémantique de cette construction est similaire à celle de la sélection. L'une des gardes ouvertes est choisie et l'éventuelle commande de communication est exécutée. Le contrôle est ensuite passé à la commande correspondante. Lorsque celle-ci termine, le contrôle revient devant l'itération. Si toutes les gardes sont fermées, l'itération termine. Dans les autres cas, le processus est mis en attente. Remarquer que, comme dans la sélection, il peut y avoir plusieurs partenaires potentiels.

1.2.3. Non déterminisme interne et externe

Considérons le système suivant

```
P_a ::  [ true -> P_b!chan(1) ; x:= 1
        ▮ true -> P_b!chan(2) ; x:= 2]
P_b ::  [ P_a?chan(y) ]
```

Exactement l'une des communications possibles a lieu, et, à la terminaison, x et y ont la même valeur. D'un point de vue antropomorphique, on peut dire que P_a a imposé son choix à P_b. La résolution du non déterminisme de P_a ne dépend que de contraintes internes à celui-ci, et en aucun cas de P_b. On parle alors de non déterminisme local ou *interne*.

Examinons maintenant le système suivant

```
P_a ::  [ P_b!chan_1() -> x:= 1
        ▮ P_b!chan_2() -> x:= 2]
P_b ::  [ P_a?chan_1() -> y:= 1
        ▮ P_a?chan_2() -> y:= 2]
```

La communication ne pouvant s'établir que sur un canal commun, x et y auront la même valeur à la terminaison. Cependant, la décision de la valeur 1 ou 2 qui serait en définitive choisie n'est locale ni à P_a ni à P_b. La décision appartient en fait au système de communication sous-jacent (souvent appelé "éther"). Cette forme de non déterminisme est appelée *externe* . Considérons maintenant la variante suivante du système ci-dessus où les commandes de communications de P_a ont été placées en dehors des gardes.

```
P_a ::  [ true -> P_b!chan_1() ; x:= 1
▮       true -> P_b!chan_2() ; x:= 2]
P_b ::  [ P_a?chan_1() -> y:= 1
▮       P_a?chan_2() -> y:= 2]
```

Ce système a *exactement* les mêmes comportement que le précédent. Cependant, le non déterminisme de P_a est maintenant interne. P_a peut faire librement le chox de la valeur finale, 1 ou 2. Le non déterminisme de $Psubb$ est par contre externe. P_b ne peut imposer son choix à $Psuba$. Considérons maintenant le cas où P_a et P_b peuvent prendre localement une décision.

```
Pa :: [ true -> Pb!chan1() ; x:= 1
      ▮ true -> Pb!chan2() ; x:= 2]
Pb :: [ true -> Pa?chan1() ; y:= 2
▮     true -> Pa?chan2() ; y:= 2]
```

Ici, P_a et P_b disposent d'un non déterminisme interne. Leur choix est donc autonome. Si les choix sont compatibles, le système termine comme précédamment (correction partielle). Par contre, si P_a choisit l'un des alternants tandis que P_b choisit l'autre, un interblocage ("deadlock") se produit conduisant à une attente infinie des deux processus.

La combinaison des gardes d'émission et de réception permet donc de décrire des formes symétriques de négociation qui ne pourraient être décrites autrement. Nous aurons par exemple l'occasion au chapitre 7 d'étudier longuement le système d'élection symétrique en $CSP_{i/o}$ suivant.

```
Pa :: [ Pb!pebble() -> decision:= a
      ▮ Pb?pebble() -> decision:= b]
Pb :: [ Pa!pebble() -> decision:= b
      ▮ Pa?pebble() -> decision:= a]
```

Les deux processus se mettront d'accord (sous l'arbitrage de l'éther) sur le nom de l'un d'entre eux sans qu'aucun ne soit privilégié dans cette négociation. Nous montrerons au chapitre 7 que la possibilité d'une telle négociation symétrique dépend crucialement de l'utilisation conjointe des gardes d'émission et de réception.

L'exemple suivant illustre la synchronisation de N processus P_0, P_1, ..., P_{N-1} (les indices sont pris modulo N).

```
Pi :: [ left:= false ; right:= false ;
        *[ ¬left; Pi-1?pebble() -> left:= true
        ▮ ¬right; Pi+1!pebble() -> right:= true
        ]]
```

Remarquer que l'ordre des échanges est totalement non déterministe, et que tous les processus ont des rôles équivalents (ce point sera traité longuement au chapitre 4).

1.2.4. Puissance expressive

Le mécanisme des gardes de communication combiné avec le synchronisme des communications est très puissant. Les exemples ci-dessous en illustrent plusieurs aspects.

Soit G_2 le réseau

$$a \longleftarrow b$$

Considérons le système suivant construit sur G_2.

```
Pa :: [ true -> x:= 1 ; Pb?chan1()
      ▮ true -> x:= 2 ; Pb?chan2()]
Pb :: [ Pa!chan1() -> y:= 1
      ▮ Pb!chan2() -> y:= 2]
```

P_a choisit une valeur, 1 ou 2, et P_b apprend cette valeur, si bien que x et y ont la même valeur à la terminaison. L'effet net est que de l'information a pu être transmise du récepteur vers l'émetteur, en sens inverse de la communication! Ce phénomène est caractéristique de la puissance de la communication synchrone.

Dans certains cas, il peut même y avoir transfert d'information du fait même de l'absence de communication!

```
Pa :: [ Pb?chan() -> x:= 1
      ▌ true -> x:= 2]
Pb :: [ Pa!chan() -> y:= 1
      ▌ true -> y:= 2]
```

Dans ce cas, il y a plutôt échange d'information que transfert. Chacun des processus a la possibilité de choisir la valeur 2. Si l'un le fait, l'autre sera finalement forcé de faire aussi ce même choix. Si aucun des processus ne "veut" de cette valeur, la valeur 1 sera finalement choisie. Remarquer que, dans le premier cas, aucune communication n'a lieu. L'impossibilité de communiquer combinée à la possibilité constante de choisir l'autre alternant conduit au choix de la valeur 2. L'absence de communication est alors une information en soi.

Ce point illustre le rôle spécial joué par la terminaison d'un processus en *CSP*. Même si la terminaison est un événement local à un processus et invisible aux autres, elle peut influer sur le comportement de ceux-ci de manière indirecte. Ceci peut être vu en répétant deux fois les processus ci-dessus, les valeurs étant échangées.

```
Pa :: [ [ Pb?chan() -> x:= 1
        ▌ true -> x:= 2] ;
        [ Pb?chan() -> x:= 2
        ▌ true -> x:= 1]
      ]
Pb :: [ [ Pa!chan() -> y:= 1
        ▌ true -> y:= 2] ;
        [ Pa!chan() -> y:= 2
        ▌ true -> y:= 1]
      ]
```

Il n'est maintenant plus vrai que les deux procesus terminent toujours en accord. Il est possible que la première copie de P_a communique avec la seconde de P_b. Il n'y a plus la terminaison pour forcer l'accord entre les décisions de P_a et de P_b. Nous appellerons au chapitre 5 ce phénomène un manque de *synchronisation*.

Il nous faut mentionner dans ce cadre la convention de terminaison répartie de Hoare. Cette convention conduit à considérer qu'une garde de communication concernant un partenaire déjà terminé est fermée (au lieu de "entr'ouverte"). Hoare propose l'exemple d'un processus serveur

```
SERVER :: *[ USER?request
               -> answer:= compute(request) ; USER!answer
           ]
```

La terminaison de l'utilisateur provoque sous cette convention celle du serveur. Cependant, de nombreuses raisons théoriques et pratiques conduisent à ce pas considérer cette convention dans ce travail (*cf.* section 1.3.4 pour plus de détails).

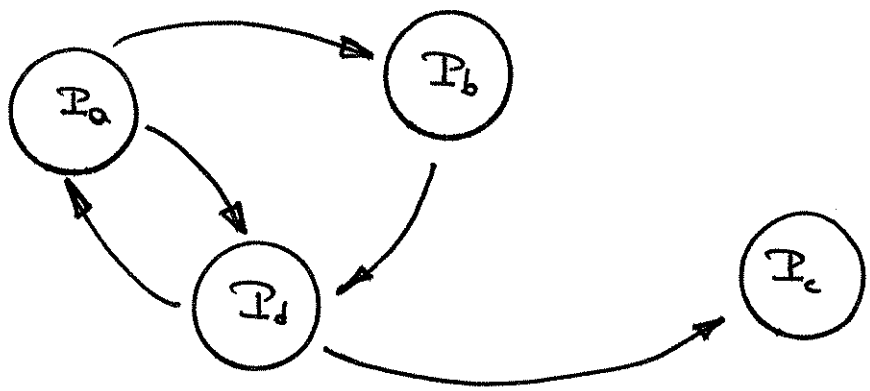


Figure 1.3.1.

1.2.5. Equité

La notion d'équité joue un rôle capital dans la sémantique des langages répartis, et nous ne ferons que l'effleurer ici. Le lecteur interessé pourra se reporter à [AFK 86] pour une récente synthèse critique des diverses approches possibles. L'ouvrage de référence est [Francez 86].

L'intuition fondamentale de l'équité est d'imposer la terminaison d'une boucle telle que

```
[ b:= true ;
  *[ b -> skip
  ▮  b -> b:= false
  ]]
```

Il devrait être clair que cette notion d'équité est implantable pratiquement en imposant un ordre sur les tests des diverses gardes. Le problème se complique lorsque l'on examine l'équité relative au comportements des processus les uns par rapport aux autres.

Dans ce travail, nous étudierons le plus souvent des systèmes n'ayant que des comportements finis. Par définition, tous ces comportements sont équitables. Nous aurons cependant besoin au chapitre 7 de la notion d'ω-equité ([Best 84]). Une transition est dite **atteignable** à partir d'un calcul (partiel) C_0 s'il existe un calcul C_1 extension de C_0 contenant cette transition. Un calcul C est dit **ω-équitable** s'il est fini, ou si toute transition atteignable à partir d'une infinité de sous-calculs de C est atteinte une infinité de fois dans C. Par exemple, si un système a une infinité de fois l'occasion d'évoluer de telle manière à terminer au cours d'un calcul ω-équitable C, alors il termine dans ce calcul. Cette notion d'équité est donc extrêmement forte. Elle n'est bien sûr pas *implantable* puisqu'elle implique la prise en compte de tous les calculs futurs possibles d'un système. Elle est cependant "acceptable" au sens de [AFK 86] (ceci montre d'ailleurs l'incomplétude des critères énoncés par ces auteurs). On définira de même la k-équité. Un calcul C est k-équitable si tout état atteignable en moins de k transitions une infinité de fois est atteint. Nous renvoyons à [Best 84] pour une discussion plus complète de ces notions.

1.3. Sémantique opérationnelle de *CSP*

Cette section décrit une sémantique opérationnelle de *CSP*, ou plutôt du sous-ensemble de *CSP* que nous utilisons dans ce travail. La présentation que nous donnons ici est largement inspirée de celle de Plotkin ([Plotkin 82]). Le cadre dans lequel se situe ce travail nous autorise plusieurs simplifications notables par rapport au travail de Plotkin. En effet, nous ne nous intéressons ici qu'aux systèmes répartis formés d'un réseau statique de processeurs séquentiels (figure 1.3.1)

Dans la terminologie de Plotkin, nous nous intéressons aux commandes de la forme

process P_a ; **process** P_b ; ...;
[$P_a::c_a$ || $P_b::c_b$ || ...]

où l'opérateur de composition parallèle || n'apparaît dans aucune des sous-commandes c_u, $u = a, b, \ldots$ (séquentialité) et où tous les noms de processeurs apparaissant dans les commandes de communication sont précisément parmi les noms P_u, $u = a, b, \ldots$ (programme clos pour les communications).

Conformément à la présentation de Plotkin, les objets syntaxiques de niveau inférieur du langage ne seront pas analysés. En effet, l'intérêt de *CSP* réside essentiellement dans le traitement du non déterminisme, du parallélisme et de la communication qu'il propose, et non dans la structure de ses expressions arithmétiques! *CSP* sera donc présenté ici comme une extension d'un langage séquentiel sous-jacent. Le choix du langage est évidemment secondaire. Conformément au travail de Richier ([Richier 83]), nous considèrerons le langage Pascal avec une certaine liberté cependant, notamment au niveau des déclarations ou des affectations lorsque ces abus de langage ne prêtent pas à confusion.

1.3.1. Sémantique du niveau inférieur

Nous considèrerons que la sémantique des objets du langage séquentiel sous-jacent est donnée en terme de système de transitions, c'est-à-dire d'un ensemble de configurations ou états, et de relations de transitions entre ces états. On notera classiquement

$$s \longrightarrow^{\lambda} s'$$

l'existence d'une transition λ d'état initial s et d'état final s'. On étendra cette notation par la convention suivante

$$s \longrightarrow^{\lambda_1|\lambda_2} s'_1|s'_2$$

qui signifie que, à partir de l'état s, une transition λ_1 est possible, conduisant à l'état s'_1, et une autre transition λ_2 conduit à l'état s'_2. Intuitivement, une transition correspond à un pas atomique de calcul pour l'objet considéré. L'état de l'objet entre l'état initial et l'état final est considéré comme inaccessible à l'observation.

Nous devons maintenant définir l'ensemble des configurations que nous considérons. Nous supposons donné dans tout ce travail un ensemble (infini) Var de variables x, y *etc.* et un ensemble Val de valeurs v, w *etc.* Un environnement *env* est une fonction de Var dans Val. L'ensemble des environnements est noté *Env*. Les configurations seront de l'une des formes (disjointes) suivantes

1) $\langle c, env\rangle$, où c est un composant syntaxique;
2) *env*;
3) *tt* ou ff, représentant les valeurs de vérité;
4) *error*, représentant un etat où une erreur a été détectée.

Les transitions seront identifiées par leurs états initiaux et finaux, éventuellement augmentés par une étiquette dont la structure sera définie plus loin. Il est a noter que, contrairement à Plotkin, nous n'aurons pas besoin de distinguer entre les notions de "failure" et de "abortion". En effet, nous éviterons la catégorie syntaxique des commandes gardées.

A chaque constituant syntaxique c, nous associons deux sous-ensembles de Var, notés $RV(c)$ et $WV(c)$. Ces ensembles représentent respectivement les variables accédées en lecture et en écriture par le constituant c, que c soit une expression ou une instruction. Nous supposerons que les constituants syntaxiques du langage séquentiel sous-jacent satisfont les axiomes suivants. Supposons qu'une transition λ soit possible dans un environnement *env* pour le constituant c

$$\langle c, env\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | env' | error | tt | ff$$

Notons $env[v/x]$ l'environnement obtenu en donnant à la variable x la valeur v dans l'environnement *env*. On admettra pour v la valeur spéciale *undef* auquel cas la valeur de x est indéfinie dans l'environnement résultant.

Axiome 1

si $x \notin WV(c)$ alors $env'(x) = env(x)$.

Axiome 2

si $x \in WV(c)$ et $x \notin RV(c)$ alors il existe une transition

$$\langle c, env[v/x]\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | env' | error | tt | ff$$

pour tout v.

En d'autres termes, nous supposons que les variables qui ne sont pas accédées en écriture gardent leur valeur, et que la valeur finale de celles accédées en écriture seulement est indépendante de leur valeur initiale.

Commandes atomiques

Nous supposons donné un ensemble *Acmd* de commandes atomiques *cmd*. L'exécution de ces commandes sera considérée comme ininterruptible lors de la mise en parallèle. Conformément aux remarques de Lamport et Schneider ([LS 84]), on peut toujours considérer une suite *finie* de transitions comme atomique sans changer la correction totale du programme. Seule la divergence doit être écartée, puisqu'une action atomique infinie ne peut avoir de sens. Dans ce travail, nous noterons donc l'atomicité d'une suite d'actions par des chevrons. Par exemple

< x:= 1 ; y:= 2 >
< $tree$:= $FIND\_SPANNING\_TREE(graph)$ >
< $decision$:= $decide(graph)$ >
< **if** $my\_name \leq new\_name$ **then** $decision$:= new_name **end** >

Chaque commande atomique est supposée décrite par un ensemble de transitions

$$\langle cmd, env\rangle \longrightarrow env' | error$$

Nous prenons en compte la possibilité d'erreurs dynamiques, détectées seulement à l'exécution, par l'introduction d'un état spécial noté *error*. Typiquement, nous aurons la transition

$$\langle x := m/n, env\rangle \longrightarrow error$$

si $env(n) = 0$. Nous ne nous intéresserons pas ici à la manière dont on passe de la forme syntaxique d'une commande *cmd* au système de transition qui la décrit. Rien n'interdit de considérer en particulier des commandes atomiques non déterministes telle que x :=? auxquelles seraient associées les transitions

$$\langle x :=?, env\rangle \longrightarrow env[n/x]$$

pour n = 0, 1, 2, ...On peut aussi envisager des commandes indéfinies sur certains environnement, telles que $\langle cmd, env\rangle$ n'admette aucune transition.

Expression booléennes

Nous supposerons donné un ensemble *Bool* d'expressions booléennes notées *bool*, par exemple (le point-virgule désigne la conjonction)

`true`

`false`

halt

$\neg done ; received[in]$

Leur sémantique est exprimée sous la forme de transitions

$$\langle bool, env \rangle \longrightarrow tt|ff$$

Ici encore, rien n'empêche de considérer des prédicats non déterministes ou partiellement définis. Remarquons que nous considérons que l'évaluation d'une expression booléenne ne peut provoquer d'erreur explicite, contrairement à Plotkin. Cette hypothèse simplifie quelque peu l'exposé et ne semble pas restrictive. Si nécessaire, on pourra procéder à une première évaluation de l'expression comme partie droite d'une affectation! Du fait de leur statut d'expression, les expressions booléennes ne modifient pas leur environnement

$$WV(bool) = \emptyset$$

Expressions de communication

Le langage *CSP* assimile une communication à une affectation répartie, le processus émetteur fournissant la valeur, et le processus récepteur la variable. De manière plus générale, Plotkin introduit les notions d'expression d'émission et d'expression de réception, qui correspondent respectivement aux parties droites et gauches d'une affectation réalisée par filtrage. Dans le cas de la réception, le mot "expression" est abusif, puisque même si cet objet a l'apparence syntaxique d'une expression, il denote cependant une modification d'environnement.

On suppose donc donné un ensemble *Output* d'expression d'émission *output*, par exemple

$(x,\ y,\ z)$

$accumulator \wedge result[in]$

( ) (dans le cas d'une synchronisation pure)

Dans un environnement donné, de telles expressions dénotent les valeurs émises par le canal considéré. Leur sémantique est donc spécifiée par un ensemble de transitions, d'états initiaux de la forme $\langle output, env \rangle$, et d'état finaux l'ensemble Val.

$$\langle output, env \rangle \longrightarrow v$$

On considère qu'une émission de valeur ne peut modifier l'environnement.

$$WV(output) = \emptyset$$

Remarquons que, là encore, nous admettons que l'évaluation D'une expression d'émission ne puisse produire d'erreur.

On suppose d'autre part donné un ensemble *Input* d'expressions de réception *input*, par exemple

$(x,\ y,\ z)$

( )

Selon l'intuition habituelle, la réception d'une valeur dans une variable n'est pas dépendante de la valeur précédente de cette variable. Nous supposons donc

$$RV(input) = \emptyset$$

Notons que ceci exclut l'utilisation d'un élément de tableau comme variable lors d'une réception. La sémantique d'une expression de reception dépend de l'environnement initial, mais aussi de la valeur reçue. Les transitions sont donc étiquetées par cette valeur. Elles sont de la forme

$$\langle input, env\rangle \longrightarrow^{v} env'$$

Ici encore, nous admettons un comportement non déterministe lors de la réception d'une valeur donnée. De plus, nous admettons qu'une expression puisse refuser une valeur, ce qui serait rendu par l'absence de transition étiquetée par cette valeur. Nous faisons l'hypothèse que la réception d'une valeur ne peut provoquer d'erreur. Cette hypothèse est de nature plus forte que précédemment à cause du caractère réparti de l'affectation. En effet, il n'est opérationnellement pas possible de déterminer si la réception d'une valeur va provoquer une erreur avant que cette réception ait effectivement eu lieu! En fait, on considère en CSP que plutôt que de provoquer une erreur, la transmission ne se produit pas. Cette approche est analogue à celle adoptée par exemple dans les bases de données réparties. Une mise à jour doit se faire sur tous les sites ou sur aucun d'entre eux. Si cette mise à jour est refusée par un des sites, on doit pouvoir considérer qu'elle n'a purement et simplement pas eu lieu. Les protocoles assurant ce type de propriété sont souvent appelés "commitment protocols". Ils se décomposent généralement en deux phases: une phase d'interrogation, puis une phase de diffusion. Les hypothèses que nous faisons ici sur la communication en CSP supposent donc implicitement l'existence d'un protocole de ce type sous-jacent à toute communication.

1.3.2. Sémantique du niveau supérieur

Après avoir défini ci-dessus les hypothèses que nous considérons sur le langage séquentiel sous-jacent à CSP, nous pouvons maintenant définir la sémantique opérationnelle de CSP proprement dit.

Dans tout ce qui suit, on considère un ensemble (infini) $Name$ d'identificateurs (noms) de processeurs a, b *etc.* On considère aussi un ensemble (infini) $Chan$ d'identificateurs de canaux de communication *chan*. Il faut remarquer ici que de très nombreux travaux sur CSP ([Hoare 78] en particulier!) considère que l'ensemble des identificateurs de processeurs (processus) est en fait l'ensemble des entiers naturels, ce qui est parfaitement légitime. Cependant, cette confusion peut considérablement modifier la puissance d'expression du langage. En effet, les processus ayant "conscience" de leur propre nom peuvent alors le comprarer avec ceux des autres processus, et définir ainsi un ordre de priorité parmi eux de manière globale. Nous serons amené à interdire ce genre d'utilisation aux chapitres suivants pour garantir la symétrie des systèmes. Les noms de processus pouvant être utilisé comme constantes dans leur propre texte, le moyen le plus naturel est d'introduire un type spécial $Name$ sur lequel seul un certain nombre d'opérations est possible. Nous reviendrons longuement sur ce problème en section 4.3.3. Conformément à l'usage, le processus a du système P sera noté P_a.

La sémantique des commandes et systèmes sera donnée sous la forme d'un système de transition engendré par des règles de la forme

$$\frac{hypothese}{conclusion}$$

ou simplement

$$conclusion$$

si l'hypothèse est vide. Nous autoriserons la forme abrégée suivante

$$\frac{hyp_1 | hyp_2 | \ldots}{concl_1 | concl_2 | \ldots}$$

pour exprimer une famille de règles conditionnelles

$$\frac{hyp_i}{concl_i}$$

avec $i = 1, 2, \ldots$ L'étiquetage des transitions permettra d'identifier l'événement atomique ayant causé cette transition.

1.3.2.1. Commandes atomiques

Nous décrivons la sémantique de l'ensemble Com des commandes atomiques c propres de CSP (à ne pas confondre avec les commandes atomiques cmd du langage sous-jacent!)

instruction vide

$$c ::= \mathtt{skip}$$

L'instruction *skip* est l'instruction vide, élément neutre de la composition séquentielle de CSP. Sa sémantique est simplement

$$\langle \mathtt{skip}, env\rangle \longrightarrow^{skip} \langle terminated, env\rangle$$

Nous avons $RV(\mathtt{skip}) = \emptyset$ et $WV(\mathtt{skip}) = \emptyset$.

instruction d'erreur

$$c ::= \mathtt{abort}$$

Cette instruction permet de produire explicitement une erreur à l'exécution.

$$\langle \mathtt{abort}, env\rangle \longrightarrow^{abort} \langle aborted, env\rangle$$

Nous avons $RV(\mathtt{abort}) = \emptyset$ et $WV(\mathtt{abort}) = \emptyset$.

instruction atomique

$$c ::= cmd$$

Une commande atomique peut aussi être constituée d'une commande atomique du langage séquentiel sous-jacent.

$$\frac{\langle cmd, env\rangle \longrightarrow env'}{\langle cmd, env\rangle \longrightarrow^{cmd} \langle terminated, env'\rangle}$$

$$\frac{\langle cmd, env\rangle \longrightarrow error}{\langle cmd, env\rangle \longrightarrow^{abort} \langle aborted, env\rangle}$$

Les ensembles de variables RV et WV sont ceux hérités du langage séquentiel sous-jacent: $RV(c) = RV(cmd)$ et $WV(c) = WV(cmd)$. Les erreurs produites au niveau du langage sous-jacent sont donc propagées au niveau de CSP. Notons qu'elles ne modifient pas l'environnement. En effet, l'atomicité implique que si l'exécution d'une commande est décidée, alors elle doit être achevée. Si cette exécution doit provoquer une erreur, elle ne peut donc avoir d'effet sur l'environnement.

instruction d'émission

$$c ::= P_u!chan(output)$$

Cette instruction provoque l'émission de la valeur résultant de l'évaluation de l'expression $output$ dans l'environnement courant sur le canal $chan$ vers le processus P_u. Rappelons que le notation P_u n'est qu'un abus dû à la tradition : il faudrait en fait écrire u.

$$\frac{\langle output, env\rangle \longrightarrow v}{\langle P_u!chan(output), env\rangle \longrightarrow^{P_u!chan(v)} \langle terminated, env\rangle}$$

On a $RV(c) = RV(output)$ et $WV(c) = WV(output) = \emptyset$. Notons l'étiquette de la transition formée de la valeur émise et du processus destinataire.

instruction de réception

$$c ::= P_u?chan(input)$$

Cette instruction spécifie la réception d'une valeur sur la canal *chan* en provenance du processus P_u. Cette valeur est reçue au travers du filtre spécifiée par l'expression de réception *input*.

$$\frac{\langle input, env\rangle \longrightarrow^{v} env'}{\langle P_u?chan(input), env\rangle \longrightarrow^{P_u?chan(v)} \langle terminated, env'\rangle}$$

On a aussi $RV(c) = RV(input) = \emptyset$ et $WV(c) = WV(output)$. On notera ici encore la forme de l'étiquette.

1.3.2.2. Structures de contrôle

Le langage *CSP* offre trois structures de contrôle locales: la composition séquentielle, l'alternative non déterministe et l'itération non déterministe.

séquencement

$$c ::= c_1;c_2$$

L'exécution de la commande c_2 suit celle de c_1 si celle-ci se termine normalement.

$$\frac{\langle c_1, env\rangle \longrightarrow^{\lambda} \langle c_1', env'\rangle | \langle terminated, env'\rangle | \langle aborted, env'\rangle}{\langle c_1;c_2, env\rangle \longrightarrow^{\lambda} \langle c_1';c_2, env'\rangle | \langle c_2, env'\rangle | \langle aborted, env'\rangle}$$

On a de plus

$$RV(c_1;c_2) = RV(c_1) \cup RV(c_2)$$

et

$$WV(c_1;c_2) = WV(c_1) \cup WV(c_2)$$

Remarquons que, classiquement, le passage du contrôle entre les deux commandes est "instantané". Ceci est exprimé grâce à l'introduction du symbole "terminated".

alternative

$$c ::= [bool_1 \Rightarrow c_1 \,\blacksquare\, \cdots \,\blacksquare\, bool_p \Rightarrow c_p]$$

Conformément à la présentation de Plotkin, nous considérons à ce niveau des commandes fortement gardées (=>) au lieu des commandes gardées de Hoare (->). Ceci permet de simplifier légèrement la présentation des gardes booléennes et des gardes contenant des commandes de communication. Les notations de Hoare sont alors vue comme des abréviations selon les règles suivantes

garde d'émission

$$bool;P_u!chan(output)\text{->}c$$

sera l'abréviation de

$$bool\text{=>}P_u!chan(output);c$$

garde de réception

$$bool;P_u?chan(input)\text{->}c$$

sera celle de

$$bool\text{=>}P_u?chan(input);c$$

garde booléenne

$$bool\text{->}c$$

sera celle de

$$bool\text{=>}\texttt{skip;}c$$

La sémantique d'une telle alternative c est décrite par les règles suivantes. Supposons que, pour un certain i,

$$\langle bool_i, env\rangle \longrightarrow tt$$

Alors,

$$\frac{\langle c_i, env\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | \langle terminated, env'\rangle | \langle aborted, env'\rangle}{\langle c, env\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | \langle terminated, env'\rangle | \langle aborted, env'\rangle}$$

Supposons que pour tout $i = 1, 2, \ldots, p$, on ait

$$\langle bool_i, env\rangle \longrightarrow ff$$

Alors,

$$\langle c, env\rangle \longrightarrow^{abort} \langle aborted, env\rangle$$

Intuitivement, ces règles peuvent s'exprimer ainsi. Nous dirons qu'une garde est **ouverte** si le booléen s'évalue à tt et si l'exécution de la commande associée peut commencer dans l'état courant. En particulier, si l'on considère une garde de communication, ceci signifie exactement que la communication est possible dans l'état courant. Nous dirons qu'une garde est **fermée** si son booléen s'évalue à ff. Sinon, elle est dite **entr'ouverte** . Lors de l'exécution d'une alternative, l'une des grades ouvertes est choisie de manière non déterministe, et l'exécution de la commande associée est initialisée. Si toutes les gardes sont fermées, une erreur se produit. Si enfin aucune garde n'est ouverte mais toutes ne sont pas fermées, rien ne se produit, et le processus reste en attente, éventuellement infinie.

Un point de sémantique doit être souligné. On considère que l'évaluation d'une garde purement booléenne constitue une transition, au lieu d'être instantanée. Ce point pourrait être discuté, car il peut sembler contraire à l'intuition. Cependant, au niveau théorique, ce choix est le plus logique, et aussi le plus simple à traiter.

On pose

$$RV(c) = RV(bool_1) \cup \ldots RV(bool_p) \cup RV(c_1) \cup \ldots \cup RV(c_p)$$

et

$$WV(c) = WV(bool_1) \cup \ldots WV(bool_p) \cup WV(c_1) \cup \cdots \cup WV(c_p)$$

itération

$$c ::= *[bool_1\text{=>}c_1 \,\blacksquare\cdots\blacksquare\, bool_p\text{=>}c_p]$$

Cette commande est en tout point similaire à la précédente. L'exécution de celle-ci est simplement itérée jusqu'à ce que toutes les gardes soient fermées. Au lieu de provoquer une erreur, une telle situation détermine donc simplement la sortie de la boucle.

Supposons que pour un certain i on ait

$$\langle bool_i, env\rangle \longrightarrow tt$$

Alors,

$$\frac{\langle c_i, env\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | \langle terminated, env'\rangle | \langle aborted, env'\rangle}{\langle c, env\rangle \longrightarrow^{\lambda} \langle c';c, env'\rangle | \langle c, env'\rangle | \langle aborted, env'\rangle}$$

Suposons que pour tout $i = 1, 2, \ldots, p$, on ait

$$\langle bool_i, env\rangle \longrightarrow ff$$

Alors,

$$\langle c, env\rangle \longrightarrow^{skip} \langle terminated, env\rangle$$

Cette dernière règle mérite quelques commentaires. Elle spécifie que la terminaison d'une itération dont toutes les gardes sont fermées n'est pas "instantanée", mais nécessite une transition *skip*, exactement comme pour l'évaluation d'une garde purement booléenne. Ici encore, l'intuition aurait sans doute préféré l'introduction d'une règle du type suivant. Si pour tout $i = 1,2,\ldots,p$

$$\langle bool_i, env\rangle \longrightarrow ff$$

alors

$$\langle c, env\rangle = \langle terminated, env\rangle$$

L'introduction de telle règle pose de nombreux problèmes théoriques, bien identifiés (mais encore mal connus) dans le cadre des systèmes de récriture. Nous préférons donc ici conserver une spécification plus simple à manipuler, même si elle ne correspond que partiellement à l'intuition que nous avons décrite en section 1.2. Cela revient en fait à considérer que

`x := 1 ; *[ false -> skip ]`

est équivalent à

`x := 1 ; skip`

plutôt qu'à

`x := 1`

Comme pour l'alternative, nous avons

$$RV(c) = RV(bool_1) \cup \ldots RV(bool_p) \cup RV(c_1) \cup \cdots \cup RV(c_p)$$

et

$$WV(c) = WV(bool_1) \cup \ldots WV(bool_p) \cup WV(c_1) \cup \ldots \cup WV(c_p)$$

1.3.3. Systèmes répartis

Nous pouvons maintenant définir la notion de système réparti en CSP. Un système réparti P est de la forme

$$P = [\ P_{u_1} :: c_1 \parallel \ldots \parallel P_{u_p} :: c_p\ ]$$

avec $p \geq 1$. Chacun des P_{u_i} est appelé *processeur*, et chacune des commandes associées c_i est appelée *processus*. En fait, on confondra souvent le processeur (la machine) et le processus (le programme).

Un système n'est défini que sous les hypothèses suivantes.

Disjonction

Pour tout processus c_i, c_j distincts,

$$(RV(c_i) \cup RV(c_j)) \cap WV(c_j) = \emptyset$$

En d'autres termes, si une variable est accédée en écriture par un processeur, elle est invisible à tous les autres processus. Par contre, une variable peut être partagée, en lecture seulement, par tous les processus. Cette condition garantit que le seul moyen pour les processus d'échanger de l'information est d'échanger des messages.

Consistance des communications

Le processus associé au processeur P_u ne peut contenir de commande de communication concernant P_u lui-même. Les seuls processeurs pouvant être nommés dans les commandes de communication sont ceux composant le système réparti.

La sémantique des systèmes répartis est donnée en terme de systèmes de transitions, de manière similaire aux commandes. Les transitions sont étiquetées par l'*ensemble* des événements atomiques les ayant produites. On aura des transitions de degré 1, correspondant à l'occurrence d'un événement local à un processus, et des transitions de degré 2, correspondant à une communication synchrone entre deux processus. En fait, il n'y aurait aucun obstacle théorique à considérer des transitions d'ordre supérieur, conformément par exemple à la sémantique de TCSP ([HBR 84]). En particulier, une synchronisation générale des tous les processus ("atomic braodcast") serait une opération extrêmement intéressante dans ce cadre.

Soit P un système réparti

$P = [\ P_a :: c_a \parallel P_b :: c_b \parallel \ldots]$

Nous avons les règles de sémantique opérationnelle suivantes.

Transition de degré 1

Supposons que l'on ait la transition suivante au niveau inférieur

$$\langle c_u, env\rangle \longrightarrow^{\lambda} \langle c', env'\rangle | \langle terminated, env'\rangle | \langle aborted, env'\rangle$$

Alors on en déduit la transition suivante au niveau supérieur

$$\langle P, env\rangle \longrightarrow^{\{P_u::\lambda\}} \langle P', env'\rangle$$

avec

$$P' = [P_a :: c'_a \parallel P_b :: c'_b \parallel \cdots]$$

et

1) $c'_v = c_v$ si $u \neq v$
2) $c'_u = c' | terminated | aborted$

Transition de degré 2

Supposons qu'au niveau inférieur on ait les *deux* transitions suivantes avec $u \neq v$

$$\langle c_u, env\rangle \longrightarrow^{P_v?chan(val)} \langle c''_u, env''_u\rangle | \langle terminated, env''_u\rangle | \langle aborted, env''_u\rangle$$

et

$$\langle c_v, env\rangle \longrightarrow^{P_u!chan(val)} \langle c''_v, env''_v\rangle | \langle terminated, env''_v\rangle | \langle aborted, env''_v\rangle$$

Alors on en déduit la transition suivante au niveau supérieur

$$\langle P, env\rangle \longrightarrow^{\{P_u::P_v?chan(val)\ ,\ P_v::P_u!chan(val)\}} \langle P', env'\rangle$$

avec

$$P' :: [P_a :: c'_a \parallel P_b :: c'_b \parallel \cdots]$$

et

1) $c'_w = c_w$ si $w \neq u, v$
2) $c'_u = c''_u | terminated | aborted$
3) $c'_v = c''_v | terminated | aborted$

et où $env'(x)$ vaut

1) $env''_u(x)$ si $x \in WV(c_u)$,
2) $env''_v(x)$ si $x \in WV(c_v)$,
3) $env(x)$ sinon.

Dans la cas d'une synchronisation multiple (ici de degré 2), les processus concernés progressent en même temps. l'environnement résultant est obtenu par combinaison des mises-à-jour effectuées par chacun des processus. La condition de disjonction implique

$$WV(c_u) \cap WV(c_v) = \emptyset$$

ce qui garantit la consistance de cette recombinaison.

1.3.4. Convention de terminaison répartie

Bien que nous ne considérions pas dans ce travail la convention de terminaison répartie introduite par Hoare pour CSP, il est utile d'en donner ici une sémantique. Cette convention exprime qu'une garde de communication addressant un processus déjà terminé (normalement ou anormalement) doit être considérée comme fermée.

considérons un système P

$$P = [\ P_a::c_a \parallel P_b::c_b \parallel \ldots]$$

dans un environnement *env*. Supposons que c_u soit de la forme

```
[ guard_1 -> c_1
▌ ...
▌ guard_p -> c_p ]
```

Supposons que pour tout $i = 1, \ldots, p$ l'on soit en présence de l'un des cas suivants.

cas 1

$$guard_i = bool_i$$

et

$$\langle bool_i, env_i \rangle \longrightarrow ff$$

cas 2

$$guard_i = bool_i; P_v?chan(input)$$

et l'on se trouve dans l'un des cas suivants.

1) $\langle bool_i, env \rangle \longrightarrow ff$
2) $c_v = terminated$
3) $c_v = aborted$

cas 3

$$guard_i = bool_i; P_v!chan(output)$$

et l'on se trouve dans l'un des cas suivants.

1) $\langle bool_i, env \rangle \longrightarrow ff$
2) $c_v = terminated$
3) $c_v = aborted$

Alors P admet la transition suivante

$$\langle P, env \rangle \longrightarrow^{\{P_u::abort\}} \langle P', env' \rangle$$

avec

$$P' = [\ P_a::c'_a \parallel P_b::c'_b \parallel \ldots]$$

et

1) $c'_v = c_v$ si $v \neq u$
2) $c'_u = aborted$

De même, avec cette fois c_u de la forme

$*[\ guard_1\ \texttt{->}\ c_1\ \blacksquare\ \ldots \blacksquare\ guard_p\ \texttt{->}\ c_p\ ]\ ;\ c'$

Sous les mêmes hypothèeses, P admet la transition

$$\langle P, env\rangle \longrightarrow^{\{P_u::skip\}} \langle P', env'\rangle$$

avec

$P' = [\ P_a::c'_a \parallel P_b::c'_b \parallel \ldots]$

et
1) $c'_v = c_v$ si $v \neq u$,
2) $c'_u = c'$ si c' est présent,
3) $c'_u = terminated$ si c' est en fait absent.

Il faut remarquer que ces règles ne respectent pas la structuration "bottom-up" que nous avons choisie dans cette présentation de la sémantique opérationnelle de CSP. Cette anomalie reflète le caractère "aberrant" de cette convention par rapport à la philosophie générale de CSP et, plus généralement, des systèmes répartis. En effet, un processus est conditionné par l'état d'un autre processus, sans qu'il y ait de synchronisation attachée à cet état. Par exemple, le système

$P = [\ P_a::\texttt{skip} \parallel P_b::\ *[\ P_a?x\ \texttt{->}\ \texttt{skip}\ ]]$

pourra terminer, alors que le système

$P = [\ P_a::P_b?y \parallel P_b::\ *[\ P_a?x\ \texttt{->}\ \texttt{skip}\ ]]$

ne le pourra pas. Cependant, dans les deux cas, le processus P_b dispose exactement de la même information sur P_a. Il y a donc transmission implicite d'information. De plus, dans le cas de la communication par messages, la transmission se fait par accord mutuel entre les partenaires. Par contre, dans le cas de la convention de terminaison répartie, un processus ne peut empêcher un autre processus de detecter sa terminaison pour forcer la fermeture d'une garde. Tout se passe comme si chaque processus était équipé d'une variable d'état, accessible par tous les processus en lecture de manière asynchrone. Ceci contredit donc formellement la condition de disjonction de CSP.

Par ailleurs, Apt et Francez ([AF 84]) ont montré que le transfert implicite d'information réalisé par la convention de terminaison répartie peut être simulé par des communications explicites entre processus, des faire-part de décès en quelque sorte. Cette simulation préserve la symétrie et la modularité (du moins si les gardes de sortie sont autorisées). Cette convention n'accroît donc pas la puissance d'expression de CSP. Toutes ces raisons nous confirment donc dans notre choix de ne pas la considérer dans le cadre de ce travail.

2. Propriétés des systèmes répartis

Nous avons défini au chapitre 1 la notion de système réparti en CSP, et donné une sémantique opérationnelle de ces objets. Cette section va nous permettre de définir les outils formels nécessaires aux développements présentés dans la suite de ce travail.

Nous présentons d'abord la notion de réseau et de graphe de communication.

Nous définissons ensuite la notion fondamentale de projection d'un calcul sur un processeur. Cette notion permet de définir une sémantique dénotationnelle (fonctionnelle) des sytèmes répartis. Elle permet aussi de définir la notion fondamentale d'équivalence causale de calculs. Le résultat principal reliant ces notions est que deux calculs (finis) sont causalement équivalents si et seulement si ils ont mêmes projections. Les notions de commutation, de fusion et de restriction seront les principaux outils que nous utiliserons pour les preuves de non existence des chapitre 6 et 7.

Enfin, nous présentons deux outils de construction de systèmes complexes à partir de systèmes plus simples, la composition séquentielle répartie et l'itération répartie. Ces deux outils correspondent en fait à la notion de phase, très utilisée dans le domaine de l'algorithmique répartie.

2.1. Graphe de communication

La définition statique des partenaires par les commandes de communication permet d'associer à tout système réparti un graphe de communication qui caractérise les possibilités d'échange de messages au sein de ce système.

Soit

$$P = [\ P_a::c_a \parallel P_b::c_b \parallel \ldots]$$

un système réparti. Le graphe de communication G de P a pour sommets les processeurs P_u de P. G a un côté (dirigé) de P_u vers P_v si le processus c_u associé au processeur (sommet) P_u contient *syntaxiquement* une commande d'émission de la forme $P_v!chan(output)$ vers P_v, ou si le processus c_v associé au processeur P_v contiennet une commande de réception $P_u?chan(input)$ de P_u. Il devrait être clair que les seules communications possibles au sein de P ne peuvent se faire que le long des canaux du graphe de communication G. Par contre, il se peut que certaine possibilités ne soient effectivement utilisées dans aucun calcul de P.

Le graphe de communication G d'un système P est, d'un point de vue théorique, un unigraphe (ou monographe, ou 1-graphe) dirigé sans auto-boucle ([Berge 73]): G a au plus un seul côté dirigé d'un sommet à l'autre; d'après les conditions de consistance des communications de la section 1.3.3, un processus ne peut tenter de s'envoyer un message à lui-même. Tous les graphes considérés dans ce travail seront de ce type.

On pourrait remarquer que le graphe de communication défini ici constitue une grossière approximation de l'ensemble des synchronisations effectivement réalisées lors des calculs. La détermination exacte de cet ensemble est bien évidemment un problème indécidable, mais des approximations très fines peuvent en être données (on se reportera à un article récent de Beaten, Bergstra et Klop [BBK 86], pour un exemple très élégant d'utilisation de telles approximations). Apt ([Apt 83]) a montré que des estimations fines permettent de réduire considérablement la complexité combinatoire des preuves de programmes parallèles. L'idée est de montrer, par une évaluation symbolique partielle du programme, que certains couples de commandes de communication ne peuvent se synchroniser. Une remarque triviale est que deux commandes ne peuvent se synchroniser que si les noms de canaux sont les mêmes, par exemple. On obtient alors l'ensemble $SYNT$ de [Apt 83].

En fait, un tel raffinement est de peu d'intérêt ici. Dans tous les problèmes que nous considérons, on suppose donné par avance un réseau G. Par "réseau", nous entendons un unigraphe dirigé sans auto-boucle, c'est-à-dire plus simplement un graphe, suivant la convention exposée ci-dessus. L'on cherche alors à construire un système réparti P sur ce réseau. Les processus composant P ont pour nom les noms des sommets de G. Le processus P_u ne peut contenir une commande d'émission vers le processus P_v que si G a

un côté de P_u vers P_v. De même, le processus P_v ne peut contenir une commande de réception de P_u que sous la même condition.

Il faut noter que ces processus ne contiennent pas nécessairement de telles commandes. On pourrait, si besoin était, rajouter des commandes de communication dans des portions inatteignables de code. Le graphe de communication de P est alors un sous-graphe (recouvrant) de G. Même si l'inclusion est stricte, toutes les propriétés de P seront étudiées par rapport au réseau donné préalablement, et non par rapport à son graphe de communication. Cependant, si le réseau n'est pas précisé, on prendra par défaut le graphe de communication.

Les sommets du réseau seront souvent appelés processeurs. La notion de réseau est bien consistante avec la sémantique opérationnelle dse systèmes répartis. Soit G un réseau, et P un système construit sur ce réseau. Supposons que P admette une transition,

$$< P, env > \longrightarrow^{\lambda} < P', env' >$$

Alors P' est aussi un système réparti construit sur G. En fait, le graphe de communication de P' est un sous-graphe de celui de P. Ceci montre en particulier que si la condition de consistance des communications de la section 1.3.3 est satisfaite au début du calcul, elle le reste tout au long de ce calcul.

Nous aurons besoin, dans les systèmes développés aux chapitres suivants, d'accéder aux graphes de communication. Ces graphes seront alors représentés par les déclarations Pascal suivantes

```
type Vert :  set of Name ;
     Edg :  set of
               record Start :  Name ;
                      End :  Name
               end;
     Digraph :  record Vertices :  Vert ;
                       Edges :  Edg
                end;
```

Nous utiliserons aussi la notion de voisinage. Soit G un réseau et P un système réparti construit sur G. Le processeur P_v est un voisin sortant de P_u dans G si G a un côté du sommet u vers le sommet v. P_u est alors un voisin entrant de P_v.

2.2. Notion de calcul

Nous avons spécifié la sémantique opérationnelle de CSP en termes de systèmes de transitions. Nous définissons dans cette section un certain nombre de notions liées à cette sémantique.

Soit P_0 un système réparti et env_0 un environnement. Un .b calcul C de P_0 à partir de env_0 est une suite, finie ou infinie

$$\langle P_0, env_0 \rangle \longrightarrow^{\lambda_0} \langle P_1, env_1 \rangle \longrightarrow^{\lambda_1} \ldots \longrightarrow \langle P_p, env_p \rangle \longrightarrow^{\lambda_p} \ldots$$

de transitions définies par la sémantique opérationnelle de CSP. La **longueur** du calcul C est le nombre de transitions successives. Les couples $\langle P_i, env_i \rangle$ sont appelés **états** et les étiquettes λ_i **transitions** ou **événements**. L'état $\langle P_0, env_0 \rangle$ est appelé état **initial**. Si C est de longueur p finie, l'état $\langle P_p, env_p \rangle$ est appelé état **final** de C. Les calculs de longueur 0 sont dits **vides** et identifiés à leur unique état. L'état final d'un calcul infini n'est pas défini. De tels calculs sont dits **divergents**.

Soit C_0 un calcul *fini* d'état initial $\langle P_0, env_0 \rangle$ et d'état final $\langle P_1, env_1 \rangle$. Soit C_1 un calcul d'état initial $\langle P_1, env_1 \rangle$. On définit la **concaténation** C des calculs C_0 et C_1,

$$C = C_0; C_1$$

comme la concaténation des suites d'etats et de transitions associées. On dit alors que C **prolonge** ou **étend** C_0. On note alors

$$C_1 = C \setminus C_0$$

Il devrait être clair que C_1 est défini de manière unique. On peut définir un **ordre partiel** sur les calculs de la manière suivante.

$$C_0 \prec C_1$$

s'il existe C tel que

$$C_1 = C_0; C$$

On dit qu'un calcul est **maximal** s'il est maximal pour cet ordre, c'est-à-dire s'il ne peut être prolongé strictement. En particulier, un calcul infini est maximal. Suivant le contexte, le terme "calcul" désignera un calcul quelconque ou spécifiquement un calcul maximal. Lorsque cet abus de langage risque d'introduire un confusion, nous préciserons explicitement qu'il s'agit d'un calcul (éventuellement) partiel, ou bien d'un calcul maximal. Les calculs de longueur 0 sont minimaux pour cet ordre. Deux calculs sont comparables si et seulement si l'un est une extension de l'autre. Toute chaîne croissante de calculs admet une borne supérieure. Réciproquement, tout calcul s'exprime comme borne supérieure d'une chaîne de calculs dans lequel le calcul de rang $i+1$ se déduit de celui de rang i par concaténation d'un calcul de longueur 1. Enfin, par le lemme de Zorn, on en déduit que tout calcul se prolonge en un calcul maximal.

Soit P un système réparti

$$P = [\ P_a :: c_a \parallel P_b :: c_b \parallel \ldots]$$

Considérons une transition

$$\langle P, env \rangle \longrightarrow^{\lambda} \langle P', env' \rangle$$

Alors P' est nécessairement de la forme

$$P = [\ P'_a :: c'_a \parallel P'_b :: c'_b \parallel \ldots]$$

où les c'_u sont soit des processus CSP, soit les symboles *terminated* ou *aborted*. Nous avons déjà remarqué plus haut que si P est construit sur un réseau G, alors P' est aussi construit sur G. Un système P est dit dans un état **terminé** ou proprement terminé si tous les c_u sont en fait le symbole *terminated*. Il est dit dans un état **en erreur** si au moins l'un des c_u est le symbole *aborted*.

Un calcul est (proprement) **terminé** s'il est fini et si son état final est proprement terminé. Ce calcul est nécessairement maximal, puisqu'aucune transition ne peut alors s'appliquer. Un calcul est **en erreur** s'il contient un état au moins en erreur. Tout prolongement de ce calcul est donc aussi en erreur. En particulier, un tel calcul ne peut se prolonger en un calcul terminé. Un calcul maximal fini qui n'est ni proprement terminé ni en erreur est dit **bloqué**.

En général, les systèmes répartis que nous considérons sont **fermés** en ce sens que toutes les variables utilisées par les processus sont supposées déclarées et définies par eux-mêmes. En ce cas, les seuls calculs que nous considèrerons sont ceux où aucune variable n'est initialisée dans l'environnement initial. Dans le cas contraire, nous parlerons de systèmes **ouverts**, et nous considèrerons les calculs produits à partir d'un ensemble donné d'environnements initiaux. On dira qu'un système P **termine** si tous ses calculs terminent proprement, que P **se bloque** si tous ses calculs se bloquent *etc.*

Il nous faut faire une remarque concernant la notion d'*erreur*. Dans sa présentation initiale, Hoare semblait considérer l'erreur comme fatale, en ce sens que tout calcul en erreur est maximal. L'erreur d'un processus peut donc déterminer le blocage d'un autre processus. Le problème est analogue à celui posé par la convention de terminaison répartie étudiée ne section 1.3.4. Un événement local à un processus influe sur le comportement d'un autre processus, malgré l'absence de transmission explicite d'information entre ces processus. C'est pourquoi nous considérons ici l'erreur comme une notion locale aux processus, exactement comme la terminaison. Ce sont cependant des événements *observables* ce qui se traduit par la stabilité des états produits: un processus en erreur reste en erreur, et un processus terminé reste terminé.

Dans ce travail, nous considèrerons de nombreuses procédures de transformation de systèmes répartis. Ces procédures devront être étudiées par rapport à une certaine équivalence sur les systèmes. Nous présentons ici une équivalence très simple, de type dénotationnel ou plutôt fonctionnel, inspiré des travaux de Apt et Olderog ([AO 84]), et, semble-t-il, déjà utilisée par Elrad et Francez ([EF 82]).

L'idée est de considérer le comportement fonctionnel des systèmes répartis, c'est-à-dire le rapport entre environnement initial et environnement final, s'il en existe. De plus, on considère que seuls les processus

ayant atteint leur état final ont une contribution significative dans l'environnement produit. Nous définissons donc le **résultat** produit par un procesus au cours d'un calcul. Soit

$$P = [P_a :: c_a \parallel P_b :: c_b \parallel \ldots]$$

un système réparti, et soit

$$\langle P, env \rangle \longrightarrow^{\lambda} \langle P', env' \rangle$$

une transition. Il devrait être clair que λ ne peut être que de l'une des formes suivantes.

1) $\lambda = \{P_u :: c\}$
2) $\lambda = \{P_u :: P_v!chan(output), P_v :: P_u?chan(input)\}$

Dans le premier cas, on dit que P_u est **actif** dans la transition λ, et, dans le second cas, que P_u et $Psubv$ sont actifs dans λ. Un processeur qui n'est pas actif dans une transition est dit **passif** ou **inactif** ("idle"). Ces notions s'étendent de manière classique aux calculs. Un processeur est actif dans un calcul s'il l'est dans au moins une des transitions de ce calcul. Les modifications de l'état d'un processus par une transition sont liés à l'activité des processeurs de la manière suivante.

Théorème 2.2.1. *Soit*

$$\langle P, env \rangle \longrightarrow^{\lambda} \langle P', env' \rangle$$

une transition avec

$$P = [\ \ldots \parallel P_u :: c_u \parallel \ldots]$$

et

$$P' = [\ \ldots \parallel P_u :: c'_u \parallel \ldots]$$

Les propriétés suivantes sont alors vérifiées.

1) $WV(c'_u) \subseteq WV(C_u)$ *et* $RV(c'_u) \subseteq RV(c_u)$.
2) Si, pour tout u, $x \notin WV(c_u)$, *alors* $env'(x) = env(x)$.
3) Si P_u *n'est pas actif dans* λ, *alors* $c'_u = c_u$ *et* $env'(x) = env(x)$ *pour tout* $x \in RV(c_u) \cup WV(c_u)$.

On remarquera que la propriété 1 montre que si la condition de disjonction de la section 1.3.3 est vérifiée au début du calcul, alors elle le reste tout au long du calcul. Il est donc possible de considérer que l'environnement *env* est en fait l'union d'environnements locaux env_u, propres à chaque processus c_u. L'environnement env_u est la restriction aux variables de $RV(c_u) \cup WV(c_u)$. Remarquons que ces environnements ne sont pas disjoints, puisqu'ils partagent éventuellement des variables accédées en lecture seulement par plusieurs processus. D'autre part, ils ne recouvrent pas nécessairement l'ensemble Var de toutes les variables. L'environnement env_u est appelé **projection** de l'environnement *env* sur le processus c_u. Il ne peut être modifié que lorsque P_u est actif.

Soit C un calcul fini de P, à partir de l'environnement *env*. Soit $\langle P', env' \rangle$ un état de C où P_u a terminé. Nous dirons alors que le **résultat** de P_u dans C est la projection de env' sur P_u. Cette notion est bien définie, puisque, une fois terminé, P_u ne peut être actif à nouveau. Nous noterons ce résultat $P_u(C)$.

Le résultat de P pour le calcul C, noté $P(C)$, sera le t-uple

$$\langle P_a :: result_a, P_b :: result_b, \ldots \rangle$$

où $result_u$ vaut (C est supposé fini!)

1) $P_u(C)$ si P_u termine dans C;
2) *error* si P_u est en erreur dans C;
3) *blocked* si P_u est bloqué dans C.

Enfin, la sémantique fonctionnelle $Sem_f(P)$ du système réparti P est définie par $Sem_f(P) =$

$$\{\langle env, P(C)\rangle \text{ où } C \text{ est un calcul fini de } P \text{ à partir de } env\}$$
$$\cup\{\langle env, divergence\rangle \text{ s'il existe un calcul infini de } P \text{ à partir de } env\}$$

Remarquons que, contrairement au blocage ou à l'erreur, la divergence est considérée comme fatale, en ce sens qu'elle masque totalement le comportement du système. Cette approche est justifiée dans le cadre fonctionnel qui est le nôtre ici, puisqu'un programme divergent ne peut jamais être observé dans un état "définitif". Cependant, une analyse plus fine de la divergence serait souhaitable dans d'autres cadres où l'on s'attacherait au comportement du programme proprement dit, plus qu'à son résultat.

Remarquons enfin que si le calcul C termine proprement, on peut confondre $P(C)$ et l'environnement final de C. Cette identification nous sera utile dans le suite de notre travail.

2.3. Equivalence causale

2.3.1. Notion de projection

Soit P un système réparti et C un calcul de P. Soit $P_u :: c_u$ un processeur de P. Nous définissons la projection $proj_u(C)$ du calcul C sur le processeur P_u comme le comportement de $Psubu$ au sein du calcul C. De manière plus précise, c'est la suite des transitions auxquelles P_u participe au sein de C, seuls les détails concernant P_u étant conservés. Soit env un environnement pour P. Nous notons env_u la restriction de env aux variables lues ou écrites par le processus c_u, soit $RV(c_u) \cup WV(c_u)$. La projection $proj_u(C)$ du calcul C sur un processeur P_u est alors définie récursivement de la manière suivante.

Calcul vide

Soit

$$C = (\langle [\ldots \parallel P_u :: c_u \parallel \ldots], env\rangle)$$

Alors

$$proj_u(C) = \langle c_u, env_u\rangle$$

Processeur inactif

Si P_u est inactif dans λ, alors soit

$$C = \langle P, env\rangle \longrightarrow^{\lambda} \langle P', env\rangle$$

avec

$P = [\ \ldots \parallel P_u :: c_u \parallel \ldots]$

et

$P' = [\ \ldots \parallel P'_u :: c'_u \parallel \ldots]$

Alors

$$proj_u(C) = \langle c_u, env_u\rangle$$

Processeur actif

Si P_u est actif dans λ, alors soit

$$P = \langle P, env \rangle \longrightarrow^{\lambda} \langle P', env \rangle$$

avec

$P = [\ \ldots \parallel P_u :: c_u \parallel \ldots]$

et

$P' = [\ \ldots \parallel P'_u :: c'_u \parallel \ldots]$

Alors

$$proj_u(C) = \langle c_u, env_u \rangle \longrightarrow^{\mu} \langle c'_u, env'_u \rangle$$

avec μ tel que $P_u :: \mu$ soit un élément de l'ensemble λ.

Induction

Si C est un calcul de longueur plus grande que 1,

$$proj_u(C; C') = proj_u(C); proj_u(C')$$

Remarquons que cette notion de projection est bien définie grâce aux conditions de disjonction vues en section 1.3.3 et au théorème 2.2.1. En particulier, dans la deuxième condition 2, $env_u = env'_u$. Ceci implique que la projection d'un calcul est bien une dérivation du système de transition associé aux commandes et décrit en section 1.3.2. D'autre part, P_u est inactif dans un calcul C si et seulement si $proj_u(C)$ est une dérivation de longueur 0.

2.3.2. Transitions concurrentes

Considérons deux transitions C_1 et C_2 successives

$$\langle P_1, env_1 \rangle \longrightarrow^{\lambda_1} \langle P_2, env_2 \rangle \longrightarrow^{\lambda_2} \langle P_3, env_3 \rangle$$

Nous dirons qu'elles sont **concurrentes** si aucun processus n'est actif à la fois dans chacune d'elles. De manière intuitive, suivant le vocabulaire de Lamport ([Lamport 78]), il n'y a aucune relation de *causalité* entre elles bien qu'elles soient ordonnées temporellement. Dans une sémantique partiellement ordonnée ou causale ([Reisig 84], [DU 84], [Best 85], [GL 85], [Gastin 86] *etc.*), ces transitions seraient incomparables pour l'ordre partiel. Dans la pratique, de telles transitions correspondent à des événements concurrents au sein d'un même système, dont l'ordre temporel d'apparition est imprévisible, puisque soumis à aucune contrainte. Par opposition, si un processus P_u (au moins) est actif dans C_1 et C_2, alors C_1 doit précéder temporellement C_2 puisque pour éxecuter la transition C_2, le processus P_u doit être dans l'état où le laisse C_1. C_2 dépend alors causalement de C_1 , et l'on parlera de transitions **séquentielles**. Seules de telles transitions sont comparables dans la sémantique partiellement ordonnée citée plus haut.

Théorème 2.3.2.1. *Soit*

$$C = C_1; C_2; C_3; C_4$$

un calcul où C_2 et C_3 sont deux transitions concurrentes. Alors il existe deux transitions C'_2 et C'_3 telles que

1) les processeurs actifs dans C_i et C'_i sont les mêmes, $i = 2, 3$;
2) C_i et C'_i ont mêmes projections sur ces processeurs, $i = 2, 3$;
3) le calcul

$$C' = C_1; C'_3; C'_2; C_4$$

est défini et a mêmes projections que C sur tous les processeurs. Les transitions C'_2 et C'_3 sont uniques.

Preuve Il suffit en fait de considérer un calcul

$$\langle P_1, env_1 \rangle \longrightarrow^{\lambda_1} \langle P_2, env_2 \rangle \longrightarrow^{\lambda_2} \langle P_3, env_3 \rangle$$

où λ_1 et λ_2 sont concurrentes, et de montrer l'existence et l'unicité de $\langle P_4, env_4 \rangle$ tel que

$$\langle P_1, env_1 \rangle \longrightarrow^{\lambda_2} \langle P_4, env_4 \rangle \longrightarrow^{\lambda_1} \langle P_3, env_3 \rangle$$

Posons

$$P_i = [P_a :: c_{i,a} \parallel P_b :: c_{i,b} \parallel \ldots]$$

pour $i = 1, 2, 3$. Si un processeur P_u est inactif dans λ_1, alors $env_{1,u} = env_{2,u}$ et $c_{1,u} = c_{2,u}$. Donc P_2 est indiscernable de P_1 de son point de vue. De même, si P_u est inactif dans λ_2, alors $env_{2,u} = env_{3,u}$ et $c_{2,u} = c_{3,u}$. P_3 est alors indisernable de P_2.

Posons

$$P_4 = [P_a :: c_{4,a} \parallel P_b :: C_{4,b} \parallel \ldots]$$

avec $c_{4,u} = c_{3,u}$ si P_u est actif dans λ_2 et $c_{4,u} = c_{1,u}$ sinon. De même, posons $env_4(x) = env_3(x)$ si

$$x \in RV(c_{3,u}) \cup VW(c_{1,u})$$

et P_u est actif dans λ_2, et $env_4(x) = env_1(x)$ sinon. Il est facile de vérifier alors que les transitions désirées sont alors valides.

Montrons l'unicité. Si P_u est inactif dans λ_1 et λ_2, alors nécessairement

$$c_{4,u} = c_{1,u} = c_{2,u} = c_{3,u}$$

et

$$env_{4,u}(x) = env_{1,u}(x) = env_{3,u}(x)$$

Si P_u est actif dans λ_1, alors il ne l'est pas dans λ_2 et donc $c_{4,u} = c_{3,u}$ et $env_{4,u}(x) = env_{3,u}(x)$. Si P_u est actif dans λ_2, alors il ne l'est pas dans λ_1 et $c_{4,u} = c_{1,u}$ et $env_{4,u}(x) = env_{1,u}(x)$. P_4 et env_4 sont donc uniquement déterminés. ∎

Du fait de l'unicité, il n'y a aucun problème à identifier C_i et C_i'. On dira donc plus simplement que deux transitions concurrentes successives peuvent être commutées au sein d'un calcul sans changer les projections. En particulier, une telle commutation ne change pas son état final s'il existe.

Remarquons que le traitement des erreurs dans le cas des exécutions parallèles est déterminant pour la validité de ce théorème. En effet, si l'on adopte le point de vue affirmant qu'une erreur dans l'un processus est immédiatement fatale pour le système entier, le théorème n'est plus valide dans le cas où la seconde transition provoque une erreur. Cette remarque montre que la seule attitude raisonnable face aux erreurs est ici de laisser les conséquences sous-spécifiées. En particulier, la mise en parallèle ne peut être une opération stricte au sens de la sémantique dénotationnelle.

2.3.3. Equivalence causale des calculs

Nous avons dit plus haut que la commutation de transitions concurrentes permettait d'abstraire l'ordonnancement temporel des calculs en un ordonnancement causal. Dans la pratique, seul cet ordonnancement causal est intéressant puisqu'il caractérise les contraintes par lesquelles le calcul a été spécifié (état final en particulier). L'ordonnancement temporel est imprévisible et doit être considéré comme contingent. Il est donc naturel de considérer que deux calculs sont équivalents dès qu'ils ont même ordonnancement causal. Pour des raisons techniques longuement développées par Reisig ([Reisig 84]) et intimement liées aux notions de justice et d'équité ([AFK 86]), nous nous restreindrons au cas d'un nombre fini de commutations. Le cas infini est beaucoup plus délicat à traiter. Ce problème mériterait d'ailleurs probablement un chapitre à lui tout seul (on pourra consulter [Best 84] pour une approche de ce problème).

Définition 2.3.3.1.

Deux calculs sont causalement équivalents s'ils se déduisent l'un de l'autre par un nombre *fini* de commutations de transitions concurrentes.

En particulier, deux calculs équivalents ont même projections. Les propriétés suivantes passent donc aux classes d'équivalence (beaucoup d'autres pourraient citées):

1) C est un calcul fini;
2) C est maximal (*resp.* terminé, bloqué, divergent);
3) l'état final de C (s'il existe);
4) toutes les formes de divergences.

On pourrait voir que les notions d'équité et de justice passent aussi au quotient. Ce ne serait pas le cas avec un nombre infini de commutations (on pourra consulter [AFK 86] pour une discussion de ce point).

2.3.4. Equivalence causale et projection

L'équivalence causale préserve donc les projections des calculs. Réciproquement, nous allons voir que les projections caractérisent les classes de calcul. En d'autres termes, un calcul d'un système réparti en CSP est entièrement défini par l'ensemble des visions locales que les processus en ont. Si seulement ces processus étaient capables, d'une manière cohérente, de mettre en commun leur connaissance locale, ils pourraient alors reconstituer le calcul, à équivalence causale près! Cette propriété extrêmement forte est au coeur de l'algorithme du "Snapshot" que nous étudierons au chapitre 8. L'objet de cet algorithme est précisément d'établir les conditions garantissant la cohérence de cette mise en commun malgré l'absence d'horloge globale et de transmissions directes ou instantanées.

En fait, c'est le nommage des processus partenaires dans les commandes de communication qui est déterminant. En Ada, par exemple, le récepteur ne connaît pas son partenaire. Considérons donc les trois tâches P, Q et R suivantes.

```
task P is begin R.please ; end P ;
task Q is begin R.please ; end Q ;
task R is please : entry ;
       begin accept please ;
             accept please ;
       end R ;
```

Soit C_1 le calcul où P et R se synchronisent, puis Q et R. Soit C_2 le calcul où Q et R se synchronisent, puis P et R. Ces calculs sont causalement distincts. Cependant, comme R ne peut savoir qui l'appelle, leurs projections sont identiques.

Théorème 2.3.4.1. *Deux calculs finis d'un système réparti en CSP ayant les mêmes projections sont causalement équivalents.*

Preuve La preuve est par récurrence sur la longueur du calcul C_0. Soient C_0 et C_1 deux calculs ayant mêmes projections. Si C_0 est le calcul vide, alors toutes ses projections sont des dérivations de longueur 0. Donc toutes celles de C_1 aussi et C_1 est le calcul vide.

Supposons donc C_0 non vide. Soit

$$\langle P, env\rangle \longrightarrow^{\lambda} \langle P', env'\rangle$$

la première transition de C_0. Supposons que ce soit une transition de degré 1 dont l'étiquette λ est de la forme $\{P_u :: c\}$. Alors la projection de C_0 sur P_u commence par une transition c. C'est donc aussi le cas pour C_1. Donc la première transition où P_u est actif dans C_1 contient $P_u{::}c$. Elle est donc de degré 1 et son étiquette μ est de la forme $\{P_u :: c\}$. Elle commute donc avec toutes les transitions qui la précèdent. Donc C_1 est de la forme

$$D; < Q, f > \longrightarrow^{\lambda} < Q', f' >; D'$$

et P_u est inactif dans D. Donc, par le théorème de commutation, C_1 est équivalent à un calcul dont la première transition est de la forme

$$< R, f > \longrightarrow^{\lambda} < R', f' >$$

Comme ce nouveau calcul est équivalent à C_1 qui a mêmes projections que C_0, $R = P$ et $f = env$. De plus, $R'_u = Q'_u$ par le théorème de commutation, et $Q'_u = P'_u$ puisque C_0 et C_1 ont mêmes projections.

Supposons maintenant que λ soit une émission de P_u vers P_v. Posons $\mu_u = P_v!chan(output)$ et $\mu_v = P_u?chan(input)$. λ est donc de la forme $\{P_u :: \mu_u,\ P_v :: \mu_v\}$. Les projections de C_0 sur P_u et P_v commencent respectivement par les transitions μ_u et μ_v. C'est donc aussi le cas pour C_1. Donc la première transition de C_1 où P_u est actif contient $P_u :: \mu_u$. C'est donc une transition de degré 2, et elle contient nécessairement aussi $P_v :: P_u?chan(output')$ pour un certain $output'$. On raisonne symétriquement avec P_v, et on en déduit que cette transition est exactement la première où P_u et P_v sont actifs et que c'est exactement λ. En raisonnant comme dans le premier cas, on en déduit que C_1 est équivalent à un calcul dont la première transition est la même que celle de C_0.

Donc, à équivalence près, C_0 et C_1 sont tous deux de la forme $C; C'_0$ et $C; C'_1$, C étant non vide . C'_0 et C'_1 sont de longueur strictement moindre que C_0 et C_1. Puisque C_0 et C_1 ont mêmes projections, c'est aussi le cas de C'_0 et C'_1. On termine alors la preuve en appliquant l'hypothèse de récurrence sur la longueur de C_0 qui est fini par hypothèse. ∎

2.3.5. Fusion et restriction

Une autre propriété importante des calculs liée à la notion de projection, est la notion de fusion. Cette propriété peut être vue comme analogue à l'existence de produits amalgamés dans certaines catégories. Le principe est le suivant. Considérons deux calculs $C_0; C_1$ et $C_0; C_2$ évoluant de manière distincte après éventuellement une partie commune C_0. Supposons que les ensembles de processeurs actifs dans C_1 et C_2 soient disjoints (C_1 et C_2 sont concurrents). Alors il est possible de fusionner ces deux calculs en un seul, réalisant à la fois l'évolution exprimée par C_1 et celle exprimée par C_2.

Théorème 2.3.5.1. *Soit $C_0; C_1$ et $C_0; C_2$ deux calculs finis tels que C_1 et C_2 soient deux calculs concurrents. Alors il existe un calcul $C_0; C_3$ ayant mêmes projections que $C_0; C_i$ sur les processus actifs dans C_i , $i = 1, 2$, et sur les processus passifs à la fois dans C_1 et dans C_2.*

Preuve Il suffit de prendre $C_3 = C_1; C'_2$, où C'_2 est obtenu en modifiant l'environnement env_u des processus P_u passifs dans C_2 et actifs dans C_1 de manière convenable. ∎

Nous appellerons le calcul $C_0; C_3$ **fusion** des calculs $C_0; C_1$ et $C_0; C_2$ au-dessus de C_0. Si C_0 est le calcul vide, nous parlerons simplement de fusion. Remarquons que ce calcul est unique à équivalence causale près par le théorème 2.3.4.1.

La propriété réciproque de la fusion est la **restriction.** Soit $C = C_0; C_1; C_2$ un calcul fini, C_1 et C_2 étant des calculs concurrents. Alors $C' = C_0; C_1$ est un calcul ayant même projections que C sur les processeurs inactifs dans C_2. En fait, la restriction est utilisée le plus souvent en conjonction avec la commutation. Soit C un calcul réalisant une partition des processeurs en deux sous-réseaux G_0 et G_1. Supposons qu'il n'y ait aucun message échangé au cours de C entre les processeurs de G_0 et ceux de G_1. Alors il existe un calcul C_0 ayant même projection que C sur les processeurs de G_0, et une projection vide sur ceux de G_1. C_0 est le calcul obtenu en "oubliant" les processeurs de G_1 dans C.

3. Modularité

Un aspect fondamental dans l'évaluation d'un système réparti est sa **modularité**. Considérons par exemple le cas d'un réseau local. Il est souhaitable que ce réseau puisse intégrer de nouveaux sites sans pour autant avoir à modifier l'ensemble des sites déjà présents dans le réseau. De manière plus précise, l'une des principales qualités d'un réseau est d'être constitué de matériels hétérogènes, conçus indépendamment les uns des autres et interchangeables. Un réseau est généralement constitué de modules qui ne dépendent que des modules avec lesquels ils sont directement en relation par le réseau. En particulier, une modification *locale* du réseau, ajout d'un nouveau serveur par exemple, ne doit pas entrainer de modification *globale* des modules. Seuls les modules directement en relation avec ce serveur devront être adaptés.

Une autre manière d'exprimer cette intuition est la suivante. Les différentes parties du réseau n'ont besoin, pour fonctionner, que de la connaissance de leur environnement local dans le réseau: typiquement, les spécifications (les noms) de leurs voisins directs dans le réseau. Si des connaissances globales sont nécessaires, celles-ci devront être acquises dynamiquement par la coopération des divers processus au cours d'une phase de reconfiguration. De ce point de vue, la modularité exprime donc le remplacement de connaissance globale implicites et "innées" par une faculté d'apprentissage explicite et coopérative de ces mêmes connaissances.

La notion de modularité que nous nous proposons de définir dans ce chapitre tente de rendre compte de cette intuition. Elle est nécessairement adaptée au type de systèmes répartis que nous considérons dans ce travail et au langage CSP, mais il nous semble que la méthode pourrait être aisément généralisée à d'autres formalismes. La définition que nous proposons, à la différence de celle proposée pour la notion de symétrie, est *syntaxique*. La modularité est une propriété du texte décrivant les systèmes, et non des comportements engendrés par ces textes.

D'autre part, la modularité est une propriété *différentielle*. Elle exprime la capacité d'adaptation des processus d'un système aux modifications de leur environnement global préservant leur environnement local. Les modifications que nous considérons ici concernent essentiellement la structure du réseau sous-jacent, et se traduisent par l'ajout ou le retrait de processeurs ou de canaux. La modularité est donc une propriété, non pas d' *un* système, mais bien d'une *famille* de systèmes, à savoir précisément chacune des variations possibles du système initial.

La modularité peut donc être vue comme l'adéquation, au sein d'une famille donnée de systèmes répartis, entre l'environnement local des processeurs dans le réseau d'une part, et les processus qui leur sont associés d'autre part. Les processus associés aux processeurs ne dépendent alors que de l'environnement local de ceux-ci. Deux processeurs ayant même environnement local dans deux systèmes d'une famille modulaire doivent être interchangeables. Nous garantiront cette condition en exigeant que ces processeurs soient associés à deux processus *syntaxiquement* identiques. On peut ainsi considérer une famille modulaire comme un ensemble de systèmes construits par l'assemblage arbitraire (mais consistant) de circuits (processus) normalisés et interchangeables, à la manière d'une machine construite par assemblage de cartes "compatibles" autour d'un bus local pré-existant.

3.1. Historique

La notion de modularité a souvent été utilisée comme critère pour évaluer la qualité d'un système réparti. Considérons, par exemple, l'article de Dijkstra, Feijen et van Gasteren ([DFG 83]), où ces auteurs décrivent un algorithme de détection de la terminaison répartie pour un anneau de machines abstraites. La détection est réalisée à l'aide d'un jeton circulant de machine en machine. Dans la comparaison de leur solution avec celle de Gouda ([Gouda 81]), ces auteurs remarquent que dans cette dernière, le nombre N de machines

needs to be available to each machine.

En d'autres termes, quoique symétrique, l'algorithme de Gouda n'est pas modulaire. Les diverses machines ont besoin d'une connaissance globale concernant la topologie du réseau sous-jacent pour exécuter l'algorithme. Si l'on rajoute une machine sur l'anneau, il faudra modifier le code executé par *toutes* les machines, et pas seulement celles placées au voisinage immédiat de la nouvelle machine, pour assurer un fonctionnement correct de l'algorithme. Une modification locale du réseau entraine donc une modification globale du système.

Par contre, l'algorithme proposé par Dijkstra, Feijen et van Gasteren est modulaire. Les processus executés par les machines ne dépendent que du nom de la machine (en fait un entier), et de celui des machines voisines. L'ajout d'une nouvelle machine ne nécessitera donc que la modification des processus executés par les deux machines voisines. On peut donc considérer que l'anneau est constitué par l'assemblage de composants normalisés, caractérisés seulement par leur nom et ceux de leurs voisins. Comme nous le verront en étudiant la symétrie de cet algorithme, il y a ici un compromis entre modularité et symétrie qui semble assez général dans ce type d'algorithme.

Pour autant que nous le sachions, la notion de modularité n'a pas encore été étudiée pour elle-même dans le cadre des systèmes répartis. Cependant, de nombreux travaux y font allusion de manière plus ou moins directe. Considérons par exemple la problème de l'élection d'un "leader" dans un anneau. Vitanyi, dans la synthèse qu'il propose des différentes solutions à ce problème ([Vitanyi 84]), souligne que l'efficacité des algorithmes dépend crucialement de leur modularité. Par exemple, l'algorithme probabiliste de [IR 81] est linéaire en moyenne si les processeurs connaissent au départ la taille de l'anneau. Le cas où cette taille est inconnue n'est pas évalué par ces auteurs. De manière générale, l'efficacité des algorithmes dépend beaucoup des connaissances dont les processeurs disposent initialement. La plupart des algorithmes sont bâtis sur le fait que les processeurs possèdent un nom unique, que ces noms sont totalement ordonnés et que l'ordre est connu de tous les autres processeurs de manière plus ou moins explicite. De manière générale, le processeur de nom maximal est choisi (*cf.* [BL 85] par exemple). De manière extrêment subtile, certains algorithmes utilisent le fait que ces noms sont des entiers pour déterminer des périodes de contentions plus longues pour les processeurs de nom petit, privilégiant ainsi, dans l'ordonnacement-même des calculs, les processeurs susceptibles d'être élus!

Les bornes inférieures décrites par Korach, Moran et Zaks ([KMZ 84]) sont soumises à des conditions très strictes quant à la modularité des algorithmes considérés.

> *The model under investigation is a network of n processors with distinct identities identity*(1), *..., identity*(*n*). *No processor knows any other processor's identity. Each processor has some communication lines connecting him [sic] to others. The processor knows the lines connected to himself, but not the identities of his neighbors.*

On remarquera en passant que la convention d'écriture *identity*(*i*) interdit d'utiliser les noms de processus dans des expressions autres que l'égalité. Cette condition est précisément l'une des conditions syntaxiques suffisantes de symétrie que nous proposons!

Le modèle dans lequel se place Dana Angluin ([Angluin 80]) est aussi très restrictif quant à la modularité. En particulier, les différents automates n'ont même pas de noms distincts. Ils sont seulement caractérisés par leur degré en tant que noeud du reseau sous-jacent.

> *[...we require] that the processors be designed without any knowledge (or only very broad knowledge) of the network they are to be used in.*

Ceci garantit que tout assemblage, consistant quant aux degrés, de tels composants normalisés aura le comportement requis.

> *[...] so that when they are embedded in any connected network and started in some configuration, they are guaranteed to accomplish the desired function.*

Il est intéressant de noter que certaines caractéristiques topologiques des réseaux ne sont pas décelables par de tels systèmes.

La modularité joue un rôle crucial dans le problème de la détection de la terminaison répartie ([Francez 80]). Pour résoudre ce problème, les processeurs doivent, d'une manière ou d'une autre, s'informer de leur état courant, et ce de manière consistante. Le moyen habituellement choisi est de définir au préalable un chemin hamiltonien du graphe (s'il en existe!), connu alors de tous les processeurs initialement. Une telle

option conduit naturellement à des solutions non modulaires en général, en ce sens que, partant d'une famille de systèmes modulaire et appliquant la transformation à chaque système de la famille, l'on n'obtient pas une famille modulaire. Cependant, si le graphe est un anneau dirigé, alors il existe un chemin hamiltonien canonique, et chaque processeur peut déterminer localement sa place dans ce chemin. C'est pourquoi la plupart des auteurs se sont placés dans ce cadre. Leurs solutions sont alors au mieux modulaires par rapport à la classe des réseaux qui sont en fait des anneaux dirigés. Cependant, certains auteurs ([Rana 83], [AR 84], [Richier 84] *etc.*) supposent que la longueur de l'anneau est initialement connue de tous les processeurs. Leurs solutions ne sont donc pas modulaires. Par contre, d'autres auteurs ne font pas cette hypothèse ([Francez 80], [DFG 83] *etc.*) et obtiennent ainsi des solutions modulaires. Nous décrirons au chapitre 8 une solution modulaire (et symétrique) par rapport à la classe des réseaux fortement connexes.

3.2. Une définition syntaxique de la modularité

Nous avons vu dans la discussion ci-dessus que la modularité exprime la capacité d'adaptation des processus d'un système aux variations de leur environnement global. Plus précisément, les processus associés aux divers processeurs du système ne doivent dépendre que de l'environnement local de ces processeurs, c'est-à-dire de leurs voisins immédiats. Un système modulaire peut être vu comme un assemblage de circuits élémentaires normalisés; tout assemblage consistant de ces circuits produit un système correct.

Il serait techniquement très difficile de définir précisément la notion de modularité pour un système donné, considéré isolément. En effet, il faudrait alors exprimer que le réseau de communication sous-jacent à ce système peut être modifié, du fait, par exemple, de la panne ou de la reconfiguration de certains processeurs. Or, en CSP, le graphe de communication est défini statiquement par les processus. Modifier le graphe, c'est aussi modifier le système.

La seule issue technique à ce problème est de définir cette notion de manière *différentielle*. Nous considérerons la modularité, non pas d'un système, mais bien d'une famille $\mathcal{P}$ de systèmes construits sur divers réseaux $\mathcal{G}$. Intuitivement, chacun de ces systèmes P est la solution d'un problème général pour un réseau particulier G donné. La famille est obtenue en considérant l'ensemble des solutions fournies pour chaque réseau considéré. On identifira cette famille avec l'algorithme général que chaque système implante dans un réseau particulier.

Il nous faut maintenant définir la notion d'environnement local d'un processeur P_u composant un système réparti P. De manière classique dans la théorie des systèmes répartis, l'environnement local d'une entité est l'ensemble des informations concernant le système auxquelles l'entité a accès directement. Dans la langage CSP, chaque processeur est nommé statiquement, et l'unicité des noms fait partie de de la définition du langage (condition de disjonction). Le nom d'un processeur fait donc partie de son environnement local. D'autre part, les commandes d'émission et de réception imposent de nommer statiquement les destinataires et destinateurs. En fait, cette restriction permet d'associer statiquement à un système un graphe de communication. Nous avons déjà eu l'occasion de souligner combien cette restriction particularise CSP vis-à-vis d'autres langages tels que CCS, TCSP ou même Occam ([Inmos 84]). Remarquons que le langage Ada ([LeVerrand 82]) fait un choix intermédiaire. Les destinataires sont nommés, alors que les destinateurs ne le sont pas. En CSP, donc, si le nom d'un processeur voisin est modifié, le texte du processus considéré devra aussi être modifié pour conserver le graphe de communication. L'environnement local d'un processeur doit donc aussi inclure le nom de ses voisins dans le graphe de communication.

Nous pouvons donc définir l'ensemble des processeurs ayant même environnement local dans deux réseaux.

Définition 3.2.1. *Soit G, G' deux réseaux. On dira qu'un sommet u a même environnement local dans G et G' si u est un sommet de G et de G', et si u a même voisins entrants et mêmes voisins sortants dans G et G'. On notera*

$$u \in G \cap G'.$$

Soient P et P' deux systèmes répartis construits respectivement sur des réseaux G et G'. Soit u un sommet ayant même environnement local dans ces deux réseaux. Du point de vue des processus P_u et P'_u attachés à ce sommet, les deux réseaux sont indiscernables. Nous demandons en conséquence que ces deux processus soient eux aussi sémantiquement indiscernables, c'est-à-dire en fait interchangeables (figure 3.2.1).

Une telle définition doit donc inclure une définition de l'équivalence sémantique de deux processus garantissant leur interchangeabilité dans certains contextes. on sait depuis les travaux de Olderog que ce

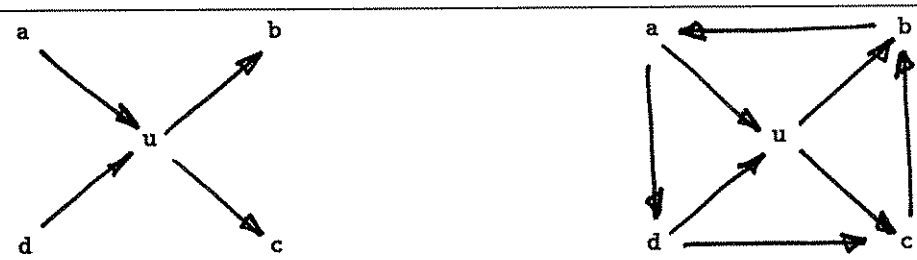


Figure 3.2.1.

type de propriété nécessite une équivalence au moins aussi forte que la "failure equivalence" (*cf.* [OH 85] pour plus de détails sur ce point). Ceci nous entrainerait dans de grandes difficultés techniques. De toute façon, nous serions amenés, de même que pour la symétrie, à considérer des conditions syntaxiques suffisantes. C'est pourquoi, dans ce travail, nous nous contenterons d'une définition syntaxique de la modularité, tout en ayant conscience de l'imperfection de ce choix.

Les deux processus P_u et P'_u devront donc avoir même texte. Ceci est possible puisque la condition sur u exprime précisément que ces deux processus ont même nom et mêmes voisins dans leur réseau respectif. Cette définition rend bien compte de notre vision intuitive des systèmes modulaires réalisés à partir de circuits élémentaires normalisés et interchangeables. On notera à ce propos que le sigle "SM90" signifie "Système Modulaire des années 1990"!

Définition 3.2.2. *Soit $\mathcal{P}$ une famille de systèmes répartis (un algorithme). Elle est dite modulaire si pour tout couple de systèmes P, P' de cette famille, construits respectivement sur les réseaux G et G', la condition suivante est satisfaite. Pour tout sommet u ayant même environnement dans G et G', le processus P_u attaché à u dans P est syntaxiquement identique au processus P'_u attaché à u dans P'.*

$$u \in G \cap G' \Rightarrow P_u = P'_u$$

Dans nos précédents travaux, cette notion avait été baptisée "généricité". En plus de l'inconvénient causé par l'introduction d'un néologisme de plus dans notre langue, il semble que le mot "modularité" convienne aussi bien à l'intuition que nous essayons d'exprimer ici.

3.3. Condition suffisante de modularité

Dans la pratique, les familles modulaires de système que nous considérons sont décrites sous la forme de maquettes. Une maquette peut être vue comme un schéma de processus paramétré par l'environnement local de celui-ci. Les systèmes de telles familles sont obtenus en combinant les processus obtenus par instanciation de la maquette. Pour garder une forme manipulable aux maquettes que nous utilisons, et pour encourager la symétrie des systèmes ainsi produits (*cf.* section 4.3.3.) nous sommes conduit à ne considérer qu'une forme restreinte de schemas de processus. Les nombreux exemples des chapitres 6, 7 et 8 montreront cependant que ce choix est un compromis acceptable entre la généralité et la complexité (*cf.* la conclusion de [Burns 81]).

Soit G un réseau et u un sommet de G. $In(u)$ est l'ensemble des voisins entrants de u, $Out(u)$ l'ensemble de ses voisins sortants. Ces deux ensembles sont parcourus respectivement par les variables *in* et *out*. Une **maquette** est un schéma de processus de la forme suivante (on omettra les arguments par la suite)

```
P_u ::
    [ Init(In, Out, u)
      *[ ▌ Guard(in, In, Out, u)
         in
            -> Command(in, In, Out, u)
         ▌ Guard(out, In, Out, u)
         out
            -> Command(out, In, Out, u)
         ▌ Guard(In, Out, u)
            -> Command(In, Out, u)
    ]]
```

où *Init* et *Command* sont des séquences de commandes atomiques ne contenant pas de commandes de communication.

Soit P_u une maquette et $\mathcal{G}$ une famille de réseaux. Soit G un réseau de $\mathcal{G}$. On obtient un système P construit sur G à partir de la maquette P_u de la manière suivante. Pour chaque sommet a de G, le processus P_a composant P est obtenu en instanciant u par a, In et Out par les ensembles $In(a)$ et $Out(a)$, et en remplaçant les commandes gardées indexées par l'ensemble des commandes gardées obtenues en substituant l'index par chacune de ses valeurs (noter que cet ensemble peut être vide). On supposera toujours que le système P ainsi obtenu est syntaxiquement correct.

Théorème 3.3.1. *Soit P_u une maquette, et $\mathcal{G}$ une famille de réseaux. Soit $\mathcal{P}$ la famille obtenue en produisant un système pour chaque réseau de $\mathcal{G}$ à partir de la maquette P_u. $\mathcal{P}$ est une famille modulaire.*

On remarquera qu'on ne fait aucune hypothèse sur les schémas *Init*, *Guard* et *Command*. En particulier, il peuvent eux-mêmes contenir des constantes de type *Name*. Par exemple, au chapitre 6, nous aurons des initialisations du type

```
if u = 0 then data:= d else data:= null end
```

Le choix de n'admettre les commandes de communication que dans les gardes de la boucle externe correspond en fait à imposer une forme normale aux processus CSP. Il est possible de montrer que tout processus CSP peut se mettre sous cette forme, au prix de l'ajout de variables auxiliaires. Une preuve partielle de ce résultat est proposée par [AC 85], mais la transformation proposée peut introduire des blocages parasites. L'importance de cette forme normale est que de tels processus CSP peuvent être vus comme des automates "event-driven" ce qui simplifie énormément leur manipulation.

4. Symétrie

Nous avons dégagé au chapitre 3 la notion de modularité, et nous en avons donné une définition dans la cadre des familles de systèmes répartis en *CSP*. La modularité peut être vue comme l'adéquation entre l'environnement local des processeurs et des processus associés à ceux-ci. Un autre point de vue est d'exprimer que les processseurs n'ont accès directement à aucune connaissance globale concernant le réseau dans lequel ils sont plongés.

La modularité est donc une notion statique. Au contraire, la symétrie est une notion concernant le comportement dynamique du système. Reprenons l'exemple du réseau local vu en section 3.0. On peut distinguer deux types de conceptions. Une conception centralisée privilégiera un certain nombre de sites chargés de gérer le réseau, et plus spécialement les diverses demandes de communication. Ces sites jouissent d'une certaine priorité par rapport aux autres sites, leur permettant ainsi de maintenir une image globale consistante du réseau servant de base à leur décisions. Souvent, il n'y a qu'un seul site de ce type, appelé serveur, moniteur ou superviseur. Dans le cas d'un réseau à jeton ([BCKKM 83]), il est par exemple spécifiquement chargé de mettre en circulation le (ou les) jetons lors de la reconfiguration du réseau. Une telle conception présente cependant un grand inconvénient. Que le superviseur tombe en panne et l'ensemble est paralysé, contrairement à la philosophie-même de la notion de réseau. De plus, la surcharge imposée au superviseur déséquilibre le fonctionnement réparti du réseau, et peut ainsi ralentir l'ensemble du réseau.

Une alternative est donc de considérer que tous les sites du réseau ont "les mêmes droits et les mêmes devoirs" dans la gestion de celui-ci. C'est par exemple le principe de fonctionnement des réseaux de type Ethernet ([SDRC 83]). Aucune priorité n'est définie entre eux, et la gestion des communications est uniformément répartie entre les sites. Ceci entraine sans aucun doute une surcharge en réalité inutile, du fait de la réplication de certains calculs sur plusieurs sites, mais cette surcharge est en tout cas uniformément répartie. De plus, la résistance aux pannes est considérablement accrue, et il n'y a plus de goulet d'étranglement au niveau du superviseur comme ci-dessus. Les différents sites se distinguent par leur nom, mais ne s'en servent pas pour déterminer un ordre de priorité. Si un tel ordre doit malgré tout être défini, ce sera par négociation globale de l'ensemble des sites. On a donc remplacé un ordre de priorité statique par une négociation explicite et coopérative de l'ensemble des sites.

La notion de symétrie a pour objet de capturer ce type d'intuition. Un algorithme est symétrique si tous les processeurs jouent des rôles équivalents dans la progression des calculs. Dans le cadre où nous nous plaçons, les capacités des processeurs sont étroitement liées à leur situation dans le réseau sous-jacent. Un processeur ayant une position centrale dans ce réseau sera naturellement appelé à jouer un rôle très différent de celui d'un processeur périphérique. La symétrie spécifie donc que des processeurs ayant des positions équivalentes dans le réseau jouent aussi des rôles équivalents dans les calculs. Elle exprime donc l'adéquation entre les caractéristiques statiques d'un réseau de processeurs, et son comportement dynamique.

La symétrie est donc une propriété liée à la *sémantique*, et non à la syntaxe comme la modularité. Son traitement formel est ainsi beaucoup délicat, et c'est sans doute ce qui explique la longueur de ce chapitre par rapport au précédent. La symétrie peut aussi être vue comme une propriété de complétude de l'espace des calculs d'un système. S'il existe un calcul privilégiant tel processeur, alors il en existe un autre privilégiant de manière équivalente tout autre processeur donné ayant une position équivalente dans le réseau. La notion de symétrie peut encore être comparée à celle d' *isotropie* en mécanique des gaz. Si à un instant donné telle particule se déplace dans telle direction, alors il aurait tout aussi bien pu se produire qu'elle se deplaçât dans toute autre direction donnée. Remarquons que cela ne signifie absolument pas que toutes les particules se déplacent à chaque instant dans toutes les directions à la fois!

Etant liée à la sémantique et non à la syntaxe, la notion de symétrie est extrêmement dépendante du cadre dans lequel nous nous plaçons. Bien que nous pensions que cette notion soit pertinente dans de nombreux domaines de l'informatique, la définition que nous en donnons est très liée à *CSP* sur les points suivants:

1) existence d'un graphe de communication dirigé, défini statiquement; pas de création dynamique ou

imbriquée de processus;
2) communications point à point, synchrones, avec nommage statique des *deux* partenaires et du canal utilisé;
3) observabilité de la terminaison et de l'erreur.

Nous essaierons tout au long de ce chapitre de souligner et d'argumenter les choix techniques que nous seront conduits à faire.

4.1. Historique

La notion de symétrie apparaît plus ou moins explicitement dans de nombreux travaux sur les systèmes répartis. Considérons de nouveau l'article de Dijkstra, Feijen et van Gasteren déjà présenté en section 3.1 ([DFG 83]). Comparant de nouveau leur solution avec celle de Gouda ([Gouda 81]), ces auteurs précisent que dans ce dernier travail

the machines [...] are treated on equal footing.

Par contre, leur algorithme fait jouer une rôle particulier à la machine appellée M_0. Il s'agit bien ici d'un problème de symétrie. la machine M_0 est amenée à se comporter de manière privilégiée par rapport aux autres machines de l'anneau, alors que rien ne la distingue dans ce même anneau. Remarquons que ce ne serait pas le cas si la détection avait lieu par exemple dans un arbre (orienté) de racine la machine M_0. Cette machine ayant alors une situation topologiquement privilégiée dans le réseau, la symétrie tolèrerait qu'elle ait aussi un comportement privilégié.

La symétrie a souvent été présentée en termes d'absence de processeur maître assurant la gestion des communications établies par les autres processeurs esclaves. La symétrie garantit donc que le système réparti se comporte effectivement de manière répartie. Aucun processeur ne jouit alors d'un contrôle global sur l'ensemble du système. Ici encore, la symétrie apparaît bien comme l'adéquation entre les caractéristiques syntaxiques d 'un système (le fait que le langage soit réparti) et ses caractéristiques sémantiques (le fait que le contrôle au sein du système soit effectivement réparti). Par exemple, l'implantation de CSP réalisée par Richier et Fauconnier ([Richier 83]) utilise un processeur spécial appelé moniteur chargé de gérer toutes les demandes de communication entre les autres processeurs. Toute émission ou réception de message se fait par appel au moniteur. En ce sens, cette implantation n'est pas symétrique.

Cohen, Lehmann et Pnueli ([CLP 83]) discutent les liens entre la symétrie des comportements et la symétrie syntaxique des processus. Ils remarquent que la seconde est "esthétiquement" souhaitable pour une présentation claire et plaisante des systèmes répartis. Cependant, la symétrie syntaxique n'a d'intérêt que si elle garantit la symétrie sémantique. Comme nous le verrons plus loin, ce n'est pas toujours le cas dans le cadre où nous nous plaçons. Ils remarquent aussi que la symétrie est une aide précieuse pour les preuves de correction. Des exemples particulièrement impressionant sont fournis par le problème des généraux byzantins traité par Toueg, Perry et Srikanth ([TPS 84]) ou Ben-Or ([BenOr 83]). Les preuves sont considérablement simplifiées, puisque par symétrie il suffit d'examiner le comportement d'un des généraux seulement. Pour s'en convaincre, on se reportera par exemple a des algorithmes analogues qui eux ne sont pas symétriques, ou plutôt sont seulement localement symétriques. La compréhension et la preuve de ces derniers algorithmes sont ainsi rendues extrêmement difficiles, puisqu'il faut distinguer de très nombreux sous-cas.

Le problème de la détection de la terminaison répartie, proposé par Francez ([Francez 80]) montre bien ce souci d'obtenir des algorithmes symétriques. Les premières solutions proposées n'étaient pas satisfaisantes à cause de l'exclusion mutuelle entre la progression du calcul originel et les vagues successives de détection ("freezing"). Ce défaut fut par la suite corrigé ([FR 82]). Cependant, un processeur fixé au préalable, disons le processeur P_0, avait la tâche d'initialiser les vagues de détection, les autres processeurs ne faisant que les propager ([FRS 81], [FR 82]). Dans d'autres types de solutions ([DS 80], [MC 82]), un processeur initiateur est chargé d'initialiser la détection à un certain point au cours du calcul, mais il se comporte ensuite de manière analogue aux autres processeurs. Ces solutions ne sont pas satisfaisantes puisque le processeur maître doit en plus de sa participation au calcul originel, gérer les vagues de détection. Il en résulte un surcroît de travail pour ce processeur qui ralentit le calcul de l'ensemble du système, contrairement au principe de la localité des interactions propre aux systèmes répartis. De plus, une panne du processeur initiateur remet en cause l'ensemble du dispositif de détection.

Toutes ces raisons conduisent à rechercher des solutions symétriques où la charge de la détection soit exactement répartie également sur tous les processeurs. Ces solutions sont dans la plupart des cas moins

efficaces que des solutions non symétriques, mais elles semblent mieux adaptées à la philosophie générale de ce type de problème. La première solution de ce type semble due à Rana ([Rana 83]). Cette solution était (bizarrement?) incorrecte sur plusieurs points cruciaux. Apt et Richier ([AR 84], [Richier 84]) ont par la suite apporté les corrections nécessaires, et ont amélioré et généralisé les idées de Rana. Dans leurs solutions, tous les processeurs peuvent initialiser la vague (commune) de détection, et tous peuvent détecter la terminaison, et initialiser alors la vague de terminaison. Nous aurons l'occasion au cours de ce travail de présenter un autre type de solution symétrique, fondé sur l'algorithme du "snapshot" de Chandy et Misra ([CM 85a]).

Dans la plupart des travaux ci-dessus, on considère une symétrie syntaxique définie souvent de manière implicite: tous les processeurs doivent exécuter syntaxiquement le même programme, aux indispensables renommages près concernant leur nom et le nom de leurs voisins immédiats dans le réseau. Dana Angluin ([Angluin 80]) a poussé plus loin la formalisation de cette idée dans le cadre des réseaux d'automates communiquant de manière synchrone. Son modèle est très proche des systèmes répartis en $CSP_{i/o}$. cependant, les automates n'ont pas de nom propre, et communiquent seulement par des canaux numérotés arbirairement. la symétrie est spécifiée en exigeant que deux automates associés à des noeuds du réseau de degrés identiques aient la même table de transition. Cette notion de symétrie est, on le constate, extrêmement restrictive.

L'inconvénient de ce type de restrictions est que la puissance d'expression du langage s'en trouve considérablement altérée. Il a été observé très tôt que la symétrie syntaxique ("textual symmetry") est une source de blocages dans les systèmes répartis ([LR 81]). Et les procédures mises en oeuvre pour éviter ces blocages conduisent à des calculs divergents. Plusieurs solutions sont dès lors envisageables. Il est possible d'introduire des concepts probabilistes permettant de négliger les blocages et divergences de probabilité nulle. Cette approche a par exemple été utilisée dans les algorithmes d'exclusion mutuelle ([LR 81] dans le cadre des "dining philosophers", et [CLP 83] dans un cadre plus général) et les problèmes de type byzantin ([BenOr 83]).

Une autre possibilité est de distinguer l'état initial du système de son comportement proprement dit ([CM 83]). Par exemple, l'algorithme de Chandy et Misra ([CM 84]) des philosophes buveurs est symétrique dans le sens suivant:

all philosophers obey precisely the same rules.

En fait, l'état initial des processeurs est supposé non symétrique (acyclique plus précisément). Cette dyssymétrie est propagée judicieusement par des moyens symétriques, ce qui permet de résoudre les conflits nés de l'application de règles semblables par plusieurs processeurs.

Une autre possiblité est de remplacer la notion de symétrie par une notion différente. Johnson et Schneider ([JS 85]) proposent la notion de *similarité*. Deux processeurs sont similaires si, dans tout calcul infini, ils entrent une infinité de fois dans le même état en même temps. Ils ne peuvent se différencier l'un de l'autre. Dans le formalisme considéré par ces auteurs, les processeurs ne sont pas nommés. En effet, deux processeurs de noms distincts peuvent trivialement se différencier. Nous trouvons que la notion de similarité proposée par ces auteurs est extrêmement restrictive, faute d'une abstraction suffisante de la notion d'état. D'autre part, la similarité est une propriété liée aux calculs considérés de manière isolée. Il nous semble que la notion de symétrie n'est pas de cette nature. La symétrie nous semble une propriété globale de l'espace des calculs d'un système, comme nous essaierons de l'exprimer en section 4.2, une sorte de propriété de complétude ce cet espace. Pour toutes ces raisons, nous trouvons que cette approche n'est pas satisfaisante quant aux buts que nous nous fixons.

Toutes ces solutions ne peuvent s'appliquer raisonnablement dans le cas de CSP. En CSP, les processeurs sont statiquement nommés. L'état initial n'a pas d'existence conceptuelle autonome dans le cas des systèmes répartis fermés que nous considérons le plus souvent ici. Une définition informelle de la symétrie textuelle peut conduire à des paradoxes. En effet, dans la notation indicée proposée par Hoare ([Hoare 78]), rien n'interdit d'utiliser les noms de processeurs dans des expressions. Dans l'exemple suivant, inspiré de l'algorithme proposé dans [FRS 81], les processeurs exécutent des processus syntaxiquement identiques. Le système résultant est pourtant grossièrement dyssymétrique.

```
Pi :: [ received:= false; sent:= false;
        init:= false; done:= false
        *[ ¬done; ¬init; P_{i+1}!pebble(i)
           -> init:= true
        ▮ ¬done; P_{i-1}?pebble(j)
           -> [ i = j -> done:= true; "je suis élu";
              ▮ f(i) ≥ f(j) -> skip
              ▮ f(i) < f(j) -> P_{i+1}!pebble(j)
              ]
        ▮ done; ¬sent; P_{i+1}!terminate()
           -> sent:= true
        ▮ ¬received; P_{i-1}?terminate()
           -> received:= true; done:= true
        ]]
```

Figure 4.1.

```
P = [ ‖ (i:0..n-1) Pi ]
Pi :: [ i = 0 -> "je suis le maître"
      ▮ i ≠ 0 -> "je suis un esclave" ]
```

Cet exemple est grossier, puisque la constante 0 apparaît explicitement dans le texte des processus. Un exemple assurément plus subtil est présenté en figure 4.1 ci-dessous. La fonction f est définie par

$$f(x) = sin(x+1)$$

Aucune constante n'apparaît dans le texte des processus. Pourtant, le processeur désigné comme maître est invariablement le même, à savoir P_i, où $f(i)$ est le maximum de $f(0), f(1), \ldots, f(n-1)$. Remarquons que i dépend bien sûr de n, mais que la constante n n'apparaît pas explicitement dans le texte! En fait, la dyssymétrie vient de l'utilisation d'un ordre total sur les noms de processeurs : P_i a priorité sur P_j si $f(i) \geq f(j)$. Cet ordre privilégie les processeurs dont le nom est supérieur pour l'ordre.

Contrairement à la structure du réseau, qui est une propriété topologique intrinsèque, les noms de processus n'ont pas d'existence propre. Ils ne servent qu'à distinguer les processus (variable muette). En particulier, deux systèmes ne différant que par les noms de leurs processus devraient intuitivement être considérés comme équivalents. C'est pourquoi l'utilisation d'un ordre total sur l'ensemble *Name* des noms de processus va, selon nous, à l'encontre de la symétrie. Il faut cependant noter que ce n'est pas l'avis de tous les spécialistes de l'algorithmique répartie. Pour prendre un exemple récent, Tan et van Leeuwen qualifient ([TL 86]) leurs algorithmes de symétriques alors qu'ils utilisent un ordre de priorité statique sur les processeurs. Il nous semble que, au contraire, seul le test d'égalité sur les noms de processeurs garantit la symétrie. Comme l'écrivent Chandy et Misra ([CM 84])

There is no priority or any other form of externally specified static partial ordering among processes.

Les exemples ci-dessus montrent que l'intuition informelle de la symétrie textuelle ne suffit pas à garantir l'absence de priorité statique. Burns ([Burns 81]) a tenté de raffiner cette notion dans ce sens. Mais, de son propre aveux, les conditions syntaxiques qu'il obtient sont

unenecessarily complex, too restrictive

Il nous semble donc que la seule issue soit de se tourner vers une définition *sémantique* de la symétrie. Les conditions syntaxiques de symétrie seront alors vues comme des conditions suffisantes, mais non nécessaires, garantissant la symétrie sémantique.

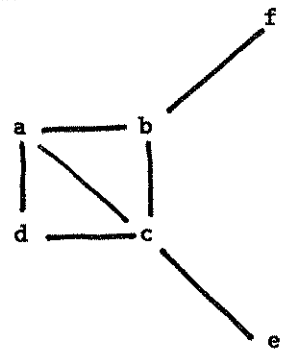


Figure 4.2.1.

4.2. Préliminaires techniques

Comme nous l'avons expliqué ci-dessus, la symétrie peut être vue comme l'adéquation entre, d'une part les capacités statiques (syntaxiques) d'un système réparti, et d'autre part ses capacités sémantiques. il nous faut donc définir des outils nous permettant de manipuler ces deux types de notions, ainsi que la notion d'adéquation.

4.2.1. Aspect statique: automorphismes de graphes

Nous avons vu en section 2.1 qu'à tout système réparti P en CSP est associé un graphe de communication ou *réseau* G. G est en fait un unigraphe dirigé sans auto-boucle. Ses sommets sont les processeurs P_u de P; G a une arête entre P_u et P_v si le processeur P_u est syntaxiquement capable d'émettre un message vers le processeur P_v ou si le processeur P_v est syntaxiquement capable de recevoir du processeur P_u.

Nous considèrerons que deux processeurs ont les mêmes capacités statiques dans le réseau sous-jacent s'ils ont des positions *topologiquement* équivalentes dans ce réseau. Ils sont alors indistinguables du point de vue de leur capacité à émettre et à recevoir des messages. Il est important de remarquer que la seule considération des liaisons directes entre le processus et ses voisins immédiats ne suffit pas pour garantir l'indiscernabilité ci-dessus. Des critères topologiques *globaux* peuvent en effet intervenir. Considérons par exemple le réseau de la figure 4.2.1.

Les processeurs P_e et P_f n'ont tous les deux qu'un seul voisin, mais P_e est le seul dont l'unique voisin ait lui-même quatre voisins. De même, P_d et P_e ont tous les deux trois voisins, mais P_d a deux voisins de degré deux, alors que P_c n'en a aucun. Ces processeurs ont donc bien des capacités statiques de communication différentes.

L'outil mathématique privilégié pour traiter ce type de propriété est la notion d' *automorphisme de graphe*. Nous rappelons ci-dessous les principales caractéristique de cet outil. Soit G un graphe orienté. Un automorphisme σ de G est une permutation des sommets de G qui préserve les côtés: si G a un côté de u vers v, alors G a un côté de $\sigma(u)$ vers $\sigma(v)$. Les automorphismes d'un graphe G forment un groupe pour la composition des applications, noté Σ_G. Σ désignera un sous-groupe de ce groupe. La permutation triviale, laissant tous les sommets invariants est notée Id_G. C'est l'élement neutre. On notera $|G|$ le nombre de sommets de G, et $|\Sigma_G|$ le cardinal de son groupe d'automorphismes. De manière évidente,

$$|\Sigma_G| \leq |G|!$$

Donc le rapport de ces deux quantité peut être vu comme une mesure de l'homogénéité du graphe G. Le graphe présenté en figure 4.2.1 n'a par exemple qu'un seul automorphisme, l'identité. Chacun de ses sommets peut être caractérisé par des considérations purement topologiques, ne faisant pas intervenir le nom de ce sommet. Par contre, le graphe complet sur n sommets a exactement $n!$ automorphimes. Réciproquement, on peut vérifier que $|\Sigma_G| = |G|!$ si et seulement si G est un graphe complet (ou alors G n'a aucun côté!)

Deux processeurs (sommets du réseau) ont des capacités (positions) équivalentes s'il existe un automorphisme σ envoyant l'un sur l'autre. L' *orbite* O_u d'un processeur P_u sous l'action d'un sous-groupe Σ

est

$$O_u = \{\sigma(u) \mid \sigma \in \Sigma\}$$

L'orbite de P_u sous l'action de Σ_G est donc l'ensemble des processeurs dont les capacités statiques sont équivalentes à celle de P_u. Nous serons aussi amené à considérer l'orbite d'un processeur sous l'action d'un automorphisme σ. Par définition, c'est son orbite sous l'action du sous-groupe de Σ_G engendré par σ. Il est facile de vérifier que les orbites des processeurs sous l'action d'un sous-groupe Σ donné forment une partition des sommets de G.

De ce point de vue, les réseaux n'ayant qu'une seule orbite apparaissent comme privilégiés puisque tous les processeurs y ont des positions équivalentes. De tels graphes sont dits *homogènes*. Par exemple, les graphes complets et les anneaux sont homogènes sous l'action de leur groupe d'automorphismes. Tout graphe G admet un sous-graphe homogène G'construit de la manière suivante. Soit a un sommet de G. Les sommets de G' sont les éléments de l'orbite de a sous l'action de Σ_G. G' a un côté entre deux sommets si et seulement si G en a un entre ces mêmes sommets. Un quotient du groupe d'automorphismes de G se plonge naturellement dans celui de G', et donc G' n'a qu'une seule orbite orbite sous l'action de $\Sigma_{G'}$.

4.2.2. Aspect dynamique: traces d'un calcul

Il nous faut maintenant définir les éléments que nous considérons comme sémantiquement signifiants dans le comportement d'un processus au cours d'un calcul donné.

Conformément à l'approche de CCS et de TCSP, nous considérons que seuls les éléments observables d'un comportement sont signifiants. L'échange d'un message entre deux processeurs sur un canal donné est habituellement considéré comme observable. Cependant, CCS ne considère pas que le contenu du message est observable. Il ne retient que l'effet de synchronisation induit par la communication. Par contre, TCSP considère que le contenu du message est observable. Il dissocie par là-même les notions de synchronisation, de communication et d'abstraction, fournissant des opérateurs distincts pour chacune des ces notions.

Dans ce travail, nous considérons que seuls les canaux de communication associés aux messages sont observables. Nous prenons donc une attitude moyenne par rapport aux deux exemples cités plus haut. En effet, ce sont les noms de canaux qui déterminent les possibilités de synchronisation entre processeurs, alors que les valeurs communiquées ne peuvent empêcher l'émission d'un message (du moins si l'utilisation d'un canal garantit, comme c'est toujours le cas ici, la validité des valeurs émises). Les parties observables d'une émission ou d'une réception pourront donc être respectivement notées

$$P_v!chan \qquad \text{et} \qquad P_v?chan$$

Nous considèrerons aussi que la terminaison et l'échec d'un processus sont observables. En particulier, ce choix implique que l'on puisse distinguer la terminaison d'un processus de sa divergence interne, ce qui n'est pas le cas en CCS où NIL est observationnellement équivalent à la solution de $X = \tau.X$. D'un point de vue opérationnel, ce type d'identification n'est pas très intéressant. En effet, la notion de *résultat* est dans ce travail fondamentale, et le propre d'un processus divergent est précisément de fournir un résultat indéfini. Il serait donc contraire à l'esprit de ce travail d'identifier des calculs terminés avec des calculs divergents, d'où la nécessité de rendre la terminaison observable. De même, nous avons introduit la notion d'échec pour prendre en compte la possibilité d'erreur survenant à l'exécution. On peut considérer qu'un processus en échec fournit un résultat spécial, un message d'erreur par exemple. Ceci le différencie radicalement d'un processus terminé qui fournit un résultat valide, et d'un processus bloqué qui ne fournit aucun résultat. Ici encore, il est nécessaire de rendre l'échec observable pour éviter l'identification de calculs en échec avec des calculs bloqués. Nous sommes donc ici en accord avec Hoare ([Hoare 84])

Certainly, successful termination should be an observable event ...

4.2.3. Notion de trace

Nous pouvons maintenant définir la trace d'un calcul sur un processeur comme la suite des actions *observables* effectuées par ce processeur au cours du calcul donné. Soit

$P = [\ P_a::c_a \parallel P_b::c_b \parallel \ldots]$

un système réparti construit sur un réseau G. Soit C un calcul de P. Les suites sont classiquement notées entre chevrons, et leur concaténation est notée par un point. Nous définissons tout d'abord une application $tr_u(C)$ qui caractérise la suite des communications auxquelles la processeur P_u participe au cours du calcul C.

Calcul vide

$$tr_u(\langle Q, env\rangle) = \langle\ \ \rangle$$

Transition locale

$$tr_u(\langle Q, env\rangle \longrightarrow^{\lambda} \langle Q', env'\rangle) = \mu$$

avec
1) $\mu = \langle\ \ \rangle$ si P_u n'est pas actif dans λ
2) $\mu = \langle skip\rangle$ si P_u est actif dans λ et λ est de degré 1

Transition de communication

$$tr_u(\langle Q, env\rangle \longrightarrow^{\lambda} \langle Q', env'\rangle) = \mu$$

avec
1) $\mu = \langle P_v!chan\rangle$ si $P_u :: P_v!chan(\ldots) \in \lambda$ (émission vers P_v)
2) $\mu = \langle P_v?chan\rangle$ si $P_u :: P_v?chan(\ldots) \in \lambda$ (réception de P_v)

Induction

$$tr_u(C'; C'') = tr_u(C').tr_u(C'')$$

Pour constituer la trace désirée, nous y adjoignons un marqueur spécifiant la terminaison ou l'échec éventuels de P_u dans C.

Cas fini

$$trace_u(C) = tr_u(C).\mu$$

si $tr_u(C)$ est fini, avec
1) $\mu = \langle terminated\rangle$ si P_u termine proprement dans C;
2) $\mu = \langle aborted\rangle$ si P_u est en échec dans C;
3) $\mu = \langle\ \ \rangle$ sinon.

Cas infini

$$trace_u = tr_u$$

si $tr_u(C)$ est infini.

Il est à noter que $trace_u(C)$ peut être finie même si le calcul C est infini. De même, un processus peut terminer properment même si d'autres divergent, se bloquent ou sont en échec. La trace d'un calcul C sur un processeur C_u sera souvent notée C_u. Les traces sont des applications monotones et continues sur l'ensemble partiellement ordonné des calculs d'un processus. Il est facile de vérifier que la trace d'un calcul sur un processeur donné est *entièrement* définie par la projection de ce calcul sur ce même processeur. Deux calculs causalement équivalents ont donc mêmes traces. La réciproque est cette fois évidemment fausse, puisque les traces "oublient" une grande partie de la sémantique des calculs.

Soit Σ_G le groupe des automorphismes de G, et soit σ un élément de ce groupe. σ agit sur les traces des calculs de P simplement en changeant tous les identificateurs de processeurs P_u en $P_{\sigma(u)}$. De manière plus précise, σ induit une application T_σ sur les traces de la manière suivante.

Calcul vide

$$T_\sigma(\langle\ \rangle) = \langle\ \rangle$$

Transition locale

$$T_\sigma(\langle skip \rangle) = \langle skip \rangle$$

Réception

$$T_\sigma(\langle P_v?chan \rangle) = \langle P_{\sigma(v)}?chan \rangle$$

Emission

$$T_\sigma(\langle P_v!chan \rangle) = \langle P_{\sigma(v)}!chan \rangle$$

Terminaison

$$T_\sigma(\langle terminated \rangle) = \langle terminated \rangle$$

Echec

$$T_\sigma(\langle aborted \rangle) = \langle aborted \rangle$$

Induction

$$T_\sigma(C'_u; C''_u) = T_\sigma(C'_u).T_\sigma(C''_u)$$

Remarquons que $T_\sigma(C_u)$ n'est pas nécessairement une trace d'un calcul de P. Cependant, cette suite d'action observable est un bon candidat pour être la trace d'un certain calcul C' sur $P_{\sigma(u)}$. En effet, si P_u peut échanger un message avec P_v, alors, par définition du graphe de communication, $P_{\sigma(u)}$ a la possibilité *syntaxique* de la faire avec $P_{\sigma(v)}$. L'application T_σ est monotone et continue sur l'ensemble partiellement ordonné des traces. En fait, l'ensemble de ces applications définit une action du groupe Σ_G sur l'ensemble des traces du système P.

4.3. Définition de la symétrie

Nous avons maintenant présenté tous les outils nécessaires pour définir précisement une notion sémantique de symétrie adaptée aux systèmes répartis en CSP. Rappelons que nous considérons que la symétrie exprime une certaine adéquation entre les caractéristiques statiques et le comportement effectif d'un système réparti P. Soit P_u un processeur de P. Supposons que l'on observe, au cours d'un calcul C de P, un certain comportement de P_u. D'après la discussion ci-dessus, ce comportement observé peut être assimilé à la trace C_u de C sur P_u. Si P est symétrique, pour tout processeur P_v dont la position est équivalente à celle de P_u dans le réseau G sous-jacent à P, un comportement analogue à C_u doit aussi être observable. Nous avons défini l'équivalence des positions par l'existence d'un automorphisme σ de G tel que $\sigma(u) = v$. Pour tout processeur $P_{\sigma(u)}$, nous demandons donc qu'il existe un certain calcul C' tel que le comportement observable de $P_{\sigma(u)}$ dans C', soit $C'_{\sigma(u)}$, soit analogue à celui de P_u dans C, soit C_u. Cette discussion nous conduit donc à la formulation suivante.

$$C'_{\sigma(u)} = T_\sigma(C_u) \qquad (4.3.1)$$

En d'autres termes, la trace de C' sur $P_{\sigma(u)}$ est la même que celle de C sur P_u, compte tenu du renommage des voisins par σ. Nous avons déjà eu l'occasion de signaler que cette condition est syntaxiquement consistante. Dans le cas où P est un système réparti fermé, C et C' ont trivialement le même environnement initial. Dans le cas de systèmes ouverts, cette exigence devra être imposée explicitement.

Remarquons que la condition (4.3.1) ci-dessus exprime une certaine forme d'isotropie. Si un phénomène amenant à privilégier une certaine direction peut se produire, alors ce phénomene pourrait tout aussi bien se produire de telle façon à privilégier n'importe quelle autre direction donnée. Ce qui évidemment ne signifie

pas que toute occurrence de ce phénomème doive privilégier *toutes* les directions à la fois! On ne demande donc pas ici que $C = C'$. Le faisant, on obtiendrait une condition plus forte que la symétrie, très proche de la notion de *similarité* proposée par [JS 85]: dans *chaque* calcul, P_u et $P_{\sigma(u)}$ sont indiscernables. Nous avons déjà discuté cette propriété en section 4.1. Il s'agit plutôt ici d'une propriété de *complétude* de l'espace des calculs: si le processeur P_u a pu avoir tel comportement dans un certain calcul, alors il existe d'autres calculs tels que les processeurs équivalents aient aussi des comportements équivalents à celui-ci.

La symétrie d'un système P peut donc être définie par l'existence pour tout automorphisme σ d'une application S_σ sur les calculs C de P telle que

$$[S_\sigma(C)]_{\sigma(u)} = T_\sigma(C_u) \tag{4.3.2}$$

Mais cette condition n'est manifestement pas suffisante. Il faut en effet que les différentes applications S_σ agissent entre elles de manière consistante sur les différents calculs. Pour la première condition, nous demanderons naturellement que les applications S_σ déterminent une action du groupe des automorphismes du graphe de communication de P.

$$(S_\sigma . S_\rho) = S_{(\sigma.\rho)} \tag{4.3.3}$$
$$S_{Id_G} = Id$$

Pour la seconde, nous demandons que chaque application S_σ respecte l'ordre partiel des calculs

$$C' \prec C'' \Longrightarrow S_\sigma(C') \prec S_\sigma(C'') \tag{4.3.4}$$

Enfin, une dernière condition est nécessaire*. Il faut en effet garantir que S_σ respecte l'équivalence causale des calculs. En d'autres termes, S_σ doit s'appliquer, non pas aux calculs intercalés, mais bien aux classes d'équivalence, c'est-à-dire en fait aux calculs partiellement ordonnés sous-jacents. La symétrie est une propriété liée à l'observation des systèmes par un observateur externe, comme le souligne la notion de trace. Or, un observateur externe ne peut distinguer les calculs causalement équivalents. Comme nous l'avons déjà remarqué, deux tels calculs ont mêmes traces. Une condition naturelle serait donc

Si C' et C'' sont causalement équivalents alors $S_\sigma(C')$ et $S_\sigma(C'')$ le sont aussi. (4.3.5)

En fait, nous aurons besoin d'une condition légèrement plus précise. remarquons tout d'abord que par (4.3.3) chaque application S_σ est en fait une bijection. Par (4.3.4), elles préservent l'ordre sur les calculs, cet ordre étant en fait celui de l'ensemble des entiers naturels. Il est facile de vérifier qu'une bijections des entiers naturels qui préserve l'ordre est en fait l'identité. On en déduit que, pour tout calcul C, C et $S_\sigma(C)$ ont même longueur. Considérons maintenant le cas d'un calcul C de la forme

$$C_1; C_2; C_3; C_4$$

Il est facile de vérifier, en utilisant (4.3.2), que $S_\sigma(C)$ est nécessairement de la forme

$$C'_1; C'_2; C'_3; C'_4$$

où chaque calcul C'_i a même longueur que le calculs C_i correspondant. De plus, toujours d'après (4.3.2), un processeur P_u est actif dans C_i si et seulement si le processeur $P_{\sigma(u)}$ est actif dans C'_i.

Supposons maintenant que les calculs C_2 et C_3 soient en fait de longueur 1, et que les ensembles de processeurs actifs dans chacun d'eux soient disjoints. D'après le théorème 2.3.2.1, ces deux transitions commutent, et l'on obtient en les transposant (avec un léger abus de notation justifié au chapitre 2)

$$C_1; C_3; C_2; C_4$$

Nous exigeons alors que l'image par S_σ de ce nouveau calcul soit précisément

$$C'_1; C'_3; C'_2; C'_4$$

En d'autres termes, les images de deux calculs se déduisant par une permutation de transitions concurrentes se déduisent l'une de l'autre par la même permutation. Remarquons que les transitions C'_2 et C'_3 sont nécessairement concurrentes d'après la propritété (4.3.2) appliquée successivement à C_1, $C_1; C_2$ et $C_1; C_3$.

Pour résumer cette discussion, nous pouvons poser la définition suivante.

* Contrairement à ce que laissait supposer une précédente version de ce travail ([Bougé 85]).

Définition 4.3.1. *Soit P un système réparti construit sur un réseau G. Soit Σ un sous-groupe du groupe des automorphismes de G. Soit $PCOMP$ l'ensemble des calculs (partiels) de G. P est dit Σ-symétrique si, pour tout $\sigma \in \Sigma$, il existe une bijection S_σ sur $PCOMP$ telle que pour tout $\sigma, \rho \in \Sigma$, $C, C' \in PCOMP$*

1)

$$\forall u \in G \quad [S_\sigma(C)]_{\sigma(u)} = T_\sigma(C_u);$$

pour tout

2)

$$S_\sigma . S_\rho = S_{(\sigma.\rho)};$$

$$S_{Id_G} = Id_{PCOMP};$$

3)

$$C \prec C' \Longrightarrow S_\sigma(C) \prec S_\sigma(C');$$

4) si C est de la forme $C_1; C_2; C_3; C_4$, alors il existe des calculs C'_1, C'_2, C'_3, C'_4 tels que

$$S_\sigma(C_1; C_2; C_3; C_4) = C'_1; C'_2; C'_3; C'_4$$

et si C_2 et C_3 sont des transitions concurrentes, alors

$$S_\sigma(C_1; C_3; C_2; C_4) = C'_1; C'_3; C'_2; C'_4$$

5) C et $S_\sigma(C)$ ont même environnement initial.

On dira plus simplement que P est symétrique s'il est Σ-symétrique pour $\Sigma = \Sigma_G$, ce qui sera le cas la plupart du temps. cependant, il est important de pouvoir aussi étudier ceratines propriétés *locales* de symétrie. Σ sera alors le groupe des automorphismes laissant invariants tous les processeurs, sauf ceux auxquels on s'intéresse. On identifiera souvent dans la suite de ce travail la bijection S_σ sur $PCOMP$ et l'automorphisme σ sur G. On notera donc $\sigma(C)$ pour $S_\sigma(C)$.

4.4. Discussion

4.4.1. Un exemple

Considérons tout d'abord l'algorithme d'élection décrit en figure 4.1. Ce système n'est pas symétrique. En effet, soit i tel que

$$f(i) = \max\{f(0), f(1), \cdots, f(n-1)\}$$

Le message envoyé par le processeur P_i est le seul à parcourir tout l'anneau avant d'être absorbé par P_i lui-même. Cette propriété est en fait une propriété des projections du calcul. Puisque les traces conservent les émissions et réceptions de messages, c'est aussi une propriété des traces de ce calcul. Si l'on considère l'automorphisme produisant une rotation de l'anneau de manière à amener $P_{(i-1)}$ sur P_i, il faudrait trouver un calcul où seul le message émis par $P_{(i-1)}$ fait un tour complet, et donc où $P_{(i-1)}$ est élu. Ceci est évidemment impossible, et donc P n'est pas symétrique. De manière plus générale, tous les algorithmes d'élection fondés sur un ordonnancement statique des processeurs (par leur nom, par exemple) ne peuvent être symétriques (*cf.* [BL 85], [TL 86] en particulier).

Au contraire, considérons le système P suivant qui sera crucial pour l'étude des systèmes électoraux symétriques.

P = [P_a ‖ P_b ‖ ...]

P_a :: [P_b?*pebble*() -> *decision*:= b
▮ P_b!*pebble*() -> *decision*:= a]

P_b :: [P_a?*pebble*() -> *decision*:= a
▮ P_a!*pebble*() -> *decision*:= b]

Le système P admet exactement quatre calculs maximaux, proprement terminés. Le premier est décrit par les transitions suivantes.

$$\langle P, env_0 \rangle \longrightarrow^{\lambda_0} \langle P_0, env_0 \rangle$$

avec

$P_0 = [\ P_a :: \ decision := a \parallel P_b :: \ decision := a\ ]$

et

$$\lambda_0 = \{P_a :: P_b!pebble(), \quad P_b :: P_a?pebble()\}$$

$$\langle P_0, env_0 \rangle \longrightarrow^{\lambda_1} \langle P_1, env_1 \rangle$$

avec

$P_1 = [P_a :: \ terminated \parallel P_b :: \ decision := a\ ]$

et

$$\lambda_1 = \{P_a :: decision := a\}$$

$$\langle P_1, env_1 \rangle \longrightarrow^{\lambda_2} < P_2, env_2 >$$

avec

$P_2 = [P_a :: \ terminated \parallel P_b :: \ terminated\ ]$

et

$$\lambda_2 = \{P_b :: decision := a\}$$

Le second calcul s'en déduit en transposant les deux dernières transitions qui sont concurrentes. Les deux autres sont obtenus en permutant a et b. Les traces de ces calculs sur P_a sont

$$\langle P_b!pebble, \quad terminated \rangle$$

et

$$\langle P_b?pebble, \quad terminated \rangle$$

Les traces sur P_b sont obtenues en remplaçant P_b par P_a, et en permutant l'émission et la réception. Le graphe de communication de P est le graphe complet sur deux sommets

$$P_a \longleftrightarrow P_b$$

Ce graphe a exactement 2 automorphismes, à savoir l'identité et la permutation des deux sommets. Ce groupe d'ordre 2 agit sur les calculs (partiels) et les traces de la manière attendue, et ce système est donc symétrique.

En fait, ce système jouit d'une propriété supplémentaire. Soit C un calcul maximal et σ un automorphisme de G. La discussion ci-dessus a montré l'existence d'un calcul $S_\sigma(C)$ tel que

$$[S_\sigma(C)]_{\sigma(u)} = T_\sigma(C_u)$$

Or, si le calcul C aboutit à l'élection de P_u, le calcul $S_\sigma(C)$ aboutit à celle de $P_{\sigma(u)}$. Un tel système sera par la suite appelé système électoral (fonctionnellement) *symétrique*.

Il n'est sans doute pas inutile de rappeler à nouveau que la symétrie est une propriété relative au graphe de communication des systèmes. Deux processeurs ayant des positions équivalentes doivent avoir des comportements équivalents. Aucune condition n'est exigée quant aux processeurs ayant des positions topologiquement privilégiées. Considérons par exemple le graphe présenté en figure 4.2.1. Tous ses sommets sont topologiquement uniques; son seul automorphisme est l'identité. Dès lors, tout système construit sur ce graphe est nécessairement symétrique! Aucune adéquation entre les aspects statiques et dynamiques n'est alors exigible. Par opposition, l'exigence de symétrie sera extrêmement forte pour des systèmes construits sur des graphes "très" symétriques, ayant un quotient $|\Sigma_G|/|G|!$ proche de l'unité.

4.4.2. Symétrie et atomicité

Nous avons longuement montré l'importance d'une définition sémantique de la symétrie, fondée sur l'espace des calculs des systèmes considérés, plutôt que sur leur forme syntaxique. Cet objectif n'est malheureurement pas atteint parfaitement par notre définition, qui est en effet extrêmement dépendante des critères d'atomicité adoptés. Considérons par exemple le système suivant

P = [P_a || P_b]
P_a :: [P_b?*pebble*() -> *decision*:= b
▮ P_b!*pebble*() -> *decision*:= a]
P_b :: [P_a?*pebble*() -> *decision*:= a
▮ P_a!*pebble*() -> *decision*:= b ; **skip**]

Ce système réparti peut être à bon droit considéré comme équivalent au système précédent. Cependant, à la différence ce ce dernier, il n'est pas symétrique. Nous avons en effet la propriété suivante

Lemme 4.4.2.1. *Soit P un système réparti, σ un automorphisme du graphe de communication de P et S_σ l'application associée sur les calculs C de P. Pour tout calcul C fini, C et $S_\sigma(C)$ ont même longueur.*

Il est alors facile de vérifier que dans tous les calculs de longueur 5, le seul message échangé est émis de P_a vers P_b, alors que ces processeurs ont des positions équivalentes. Cette remarque montre donc que notre définition reste, malgré son aspect sémantique, dépendante de la syntaxe au travers de l'atomicité des diverses actions.

Ce fait est d'autant plus regettable que l'instruction *skip* est sémantiquement vide. En particulier, le système P est (intuitivement) équivalent au système obtenu par exemple en replaçant dans P_a la commande *decision* := a par *decision* := a; *skip*. Ce nouveau système est, lui, symétrique! En fait, les conditions faisant des applications S_σ des bijections sont beaucoup trop fortes par rapport à l'intuition. Un critère analogue à celui de Milner pour l'équivalence observationnelle ce CCS serait sans doute mieux adapté, puisqu'il permet d'identifier *skip*; *skip* avec *skip*. mais nous perdrions alors le fait que ces applications déterminent une *action* du groupe Σ sur *PCOMP*, propriété qui fonde tous les résultats de non existence présentés au chapitre 7, et qui exprime l'indépendance des calculs par rapport aux noms des processeurs.

Nous nous trouvons ici devant une difficulté technique profonde, semble-t-il. L'équivalence induite par les applications S_σ est en fait analogue à l'équivalence forte de Milner. Il faudrait pouvoir l'affaiblir sans perdre l'action de Σ sur les calculs. A ce jour, nous n'avons pas de proposition valable pour réaliser un meilleur équilibre que celui proposé dans ce travail.

4.4.3. Conditions syntaxiques suffisantes

Nous avons tenté de démontrer que seule une définition sémantique de la symétrie constitue un critère acceptable pour l'évaluation des systèmes répartis. Si ce type de définition est bien adapté à la preuve de propriétés de non existence, il est très peu pratique pour vérifier qu'un système donné est symétrique. En effet, il impose de considérer l'espace de *tous* les calculs, qui est en général impossible. Une manière de contourner ce problème est de définir des propriétés *syntaxiques* suffisantes garantissant la symétrie. La difficulté est alors de trouver un compromis accepable entre d'une part la généralité et d'autre part la complexité (*cf.* la conclusion de [Burns 81] citée en section 4.1. Les conditions ci-dessous semblent réaliser un tel compromis. Elles s'appliquent à la plupart des systèmes symétriques décrits dans ce travail.

L'idée de base est bien sûr de garantir que tous les processus dérivent du même texte, aux nécessaires renommages près. Pour cela, nous reprenons la notion de *maquette* définie en section 3.3. La symétrie sera donc caractérisée par des conditions concernant la maquette, la symétrie de la maquette entrainant celle des systèmes dérivés. Intuitivement, il suffit de garantir que tous les processeurs se comportent de manière équivalente par rapport à tous leurs voisins. Il est clair, par exemple, que si aucun des composants de la maquette ne contient les symboles *In*, *Out*, *in*, *out*, ni aucune constante de nom de processeur, alors *tout* système dérivé de cette maquette est symétrique.

Soit donc P_u une maquette. Nous nous restreignons au cas où *chan* ne dépend pas des paramètres.

P_u :: [Init
*[*Bool* -> *Command*
...
▌$_{in}$ *Bool* ; P_{in}?*chan*(*Output*) -> *Command*
...
▌$_{out}$ *Bool* ; P_{out}!*chan*(*Input*) -> *Command*
]]

Soit *Env* l'espace des environnements, et *Val* delui des valeurs communicables. Soit G un réseau, et Σ un groupe d'automorphismes de G. Supposons donnée une action E de Σ sur *Env*, et une action V de Σ sur *Val*. Supposons maintenent que les différents composants de la maquette P_u valident les propriétés suivantes, pour tout sommet u de G, tout voisin entrant in de u et tout voisin sortant out de u. L'ensemble des voisins entrants de u est noté $In(u)$ et celui de ses voisins sortants $Out(u)$.

Commande atomique

Pour chaque commande atomique *cmd* de *Init* ou *Command*

$$cmd(u, In(u), Out(u), in, out)$$

posons

$$cmd_\sigma = cmd(\sigma(u), In(\sigma(u)), Out(\sigma(u)), \sigma(in), \sigma(out))$$

Alors

$$\frac{\langle cmd, env\rangle \longrightarrow env' | error}{\langle cmd_\sigma, E_\sigma(env)\rangle \longrightarrow E_\sigma(env') | error} \qquad (4.4.3.1)$$

Expression booléenne

Pour chaque expression booléenne *bool* de *Init*, *Bool* et *Command*,

$$bool(u, In(u), Out(u), in, out)$$

posons

$$bool_\sigma = bool(\sigma(u), In(\sigma(u)), Out(\sigma(u)), \sigma(in), \sigma(out))$$

Alors

$$\frac{\langle bool, env\rangle \longrightarrow tt | ff}{\langle bool_\sigma, E_\sigma(env)\rangle \longrightarrow tt | ff} \qquad (4.4.3.2)$$

Expression d'émission

Pour chaque expression d'émission *output* de *Output*

$$output = output(u, In(u), Out(u), in, out)$$

posons

$$output_\sigma = output(\sigma(u), In(\sigma(u)), Out(\sigma(u)), \sigma(in), \sigma(out))$$

Alors

$$\frac{\langle output, env\rangle \longrightarrow v}{\langle output_\sigma, E_\sigma(env)\rangle \longrightarrow V_\sigma(v)} \qquad (4.4.3.3)$$

Expression de réception

Pour chaque expression de réception *input* de *Input*

$$input = input(u, In(u), Out(u), in, out)$$

posons

$$input_\sigma = input(\sigma(u), In(\sigma(u)), Out(\sigma(u)), \sigma(in), \sigma(out))$$

Alors

$$\frac{\langle input, env\rangle \longrightarrow^v env'}{\langle input_\sigma, E_\sigma(env)\rangle \longrightarrow^{V_\sigma(v)} E_\sigma(env')} \qquad (4.4.3.4)$$

Ces conditions expriment qu'en un certain sens, l'exécution des commandes et l'évaluation des expressions booléennes ne dépendent pas, ou plutôt dépendent de manière uniforme, des noms de processeurs. Ceci proscrit, par exemple, toute commande de la forme

`if` *my_name* $\geq$ *new_name* `then` *decision*:= *my_name* `else` *decision*:= *new_name* `end`

où des noms de processeurs équivalents peuvent être comparés selon un ordre de priorité prédéfini. Par contre, le test d'égalité reste tout à fait possible. Par exemple, la commande

`if` *my_name* = *new_name* `then` *decision*:= *my_name* `end`

respecte de manière naturelle les conditions ci-dessus.

Théorème 4.4.3.1. *Soit P un système réparti construit sur un réseau G, dérivé d'une maquette satisfaisant les conditions ci-dessus, pour des actions E et V données de Σ. Si, pour tout $\sigma \in \Sigma$, E_σ préserve l'environnement initial de P, alors P est est Σ-symétrique.*

Preuve *(esquisse)* Soit C un calcul de P à partir de l'environnement initial *envsub*0. Soit σ un automorphisme de G. Nous pouvons construire $S_\sigma(C)$ à partir de C en "calquant" le comportement de $P_{\sigma(u)}$ dans $S_\sigma(C)$ sur celui de P_u dans C.

On raisonne par induction sur les calculs. Si C est un calcul d'environnement initial env_0, d'environnement final env, on construit $S_\sigma(C)$ qui est un calcul d'environnement initial $E_\sigma(env_0)$ et d'environnement final $E_\sigma(env)$. Le contrôle dans $P_{\sigma(u)}$ à la fin de $S_\sigma(C)$ est au même point que celui de P_u à la fin de C. Par les conditions (4.4.3.1), (4.4.3.2), (4.4.3.3) et (4.4.3.4), si une transition est possible pour le processus P_u, alors la transition correspondante est possible pour le processus $P_{\sigma(u)}$, et la relation entre les environnements est conservée pour les environnements résulants. En particulier, la terminaison est bien préservée. ∎

En fait, en anticipant sur les définitions du chapitre 5, nous avons le corollaire suivant.

Corollaire 4.4.3.1. *Si une maquette P_u satisfait les conditions ci-dessus alors tout système réparti construit sur cette maquette est Σ-fonctionnellement symétrique pour les actions S et E.*

Les conditions décrites ci-dessus ne sont pas à proprement parler syntaxiques. Elles sont cependant locales ce qui rend leur vérification beaucoup plus aisée. Dans la pratique, la vérification des propriétés ci-dessus est le plus souvent triviale et se réduit en fait à des considérations syntaxiques. En effet, les variables utilisées peuvent habituellement divisées en deux groupes disjoints :

1) variables de contrôle, qui permettent l'ouverture où la fermeture de gardes, et, plus généralement, la gestion du flot de contrôle;
2) variables liées au graphe de communication, construites à partir du type *Name* des noms de processeurs; par exemple les variables de type *Digraph* destinées à conserver une image du graphe de communication.

Les actions E_σ et V_σ opèrent habituellement par une simple permutation des noms de processeurs inclus dans ces variables par l'automorphisme σ considéré, laissant les valeurs scalaires invariantes. Il suffit alors que le seul tests portant sur les variables du second type soit l'égalité, et qu'il n'y ait pas d'affectation entre objets de type différent pour garantir les conditions ci-dessus.

5. Méthodes de construction de systèmes répartis

Les chapitres précédents ont dégagé les notions de modularité et de symétrie. Ils ont montré la pertinence de ces notions dans l'algorithmique répartie. Le problème qui se pose maintenant est celui de construire effectivement des algorithmes vérifiant ces propriétés.

Deux approches sont ici possibles. L'approche "compositionnelle" se propose de construire des algorithmes complexes à partir d'algorithmes élémentaires, en les composant de telle manière que certaines propriétés des composants se transmettent au résultat. Nous décrivons en section 5.1 la composition par *séquencement réparti* et *itération répartie* et étudions ses propriétés. En section 5.2, nous montrons que ce type de composition préserve la modularité et la symétrie (fonctionnelle).

Une autre approche peut être qualifiée de "transformationnelle". Le problème est de transformer un algorithme donné en algorithme équivalent, mais en vérifiant de plus certaines propriétés supplémentaires. Quand ces propriétés sont numériques, on parle dans ce cas d'optimisation. Ici, nous considérons des propriétés qualitatives, et nous parlerons plutôt d'amélioration. Nous montrons en section 5.3 que la composition par séquencement réparti permet, dans de nombreux cas, de modulariser et de symétriser les algorithmes répartis.

Il est clair qu'il est en général impossible d'améliorer tous les paramètres de qualité en même temps. La modularisation et la symétrisation se font en général au prix d'un accroissement de la complexité. Un compromis est inévitable, mais la définition de paradigmes précis permettant de le réaliser est au delà du champ de ce travail. De telles méthodes d'amélioration sont cependant importantes. Tout d'abord, elles permettent au concepteur de se concentrer sur les problèmes de correction, sachant qu'il pourra modulariser et symétriser son algorithme mécaniquement par la suite. D'autre part, les algorithmes ainsi améliorés peuvent servir de base à des optimisations plus fines, étroitement liées à la conception de l'algorithme. Les méthodes que nous proposons se situent donc sur un plan tout à fait comparable aux algorithmes de terminaison répartie tels que Francez les présente ([Francez 80]).

5.1. Composition répartie

5.1.1. Notion de phase

L'un des paradigmes de la construction de programmes, qu'ils soient centralisés ou répartis, est la notion de *phase*. Un programme P est alors vu comme une succession de phases P_1, P_2, ...qui sont elles-mêmes des programmes. Le comportement de P est d'abord identique à celui de P_1 puis, à la terminaisont de P_1 à celui de P_2 *etc.* Dans le cas centralisé, ce paradigme correspond à la composition sequentielle. Il est d'ailleurs tellement fondamental que de nombreux langages de programmation n'ont même pas d'opérateur explicite pour le dénoter, se contentant de la fin de ligne (Fortran!).

Dans le cas réparti, la mise en œuvre de ce paradigme est nettement plus complexe. Considérons deux systèmes P_1 et P_2 construits sur le même réseau. Une première approche serait de composer séquentiellement P_1 etP_2 à la manière des systèmes centralisés. Un processeur P_u ne peut donc commencer l'exécution de P_2 que lorsque tous les processeurs ont terminé celle de P_1. Nous avons déjà souligné, à propos de la convention de terminaison répartie, que ce type d'approche est contradictoire avec la philosophie-même des systèmes répartis. En effet, la terminaison d'un processus est un événement local à un processeur qui ne peut être connu d'un autre processeur.

Une approche plus conforme à la répartition est de laisser le processeur P_u commencer l'exécution de $P_{2,u}$ dès la terminaison du processus $P_{1,u}$, indépendamment de l'avancement des autres processeurs. Le passage du contrôle local entre P_1 et P_2 dans les divers pocesseurs se fait donc de manière *non synchronisée*.

Cependant, un tel comportement n'a de sens que si l'exécution de P_2 par certains processeurs ne peut interférer avec celle de P_1 par d'autres. Il faut que le calcul résultant puisse être vu comme la concaténation d'un calcul de P_1 et de P_2. En d'autres termes, il faut que tout se passe comme si l'état final de P_1 avait

virtuellement été atteint par P, même s'il n'a pas réellement atteint. Au chapitre 8, nous appellerons un tel état un "snapshot" du système P. Il est impossible, pour les composants du système, de discerner si cet état a été atteint ou non. Considérons l'exemple suivant.

$c_{1,a}$ = [P_b?*pebble*() -> $done_{1,a}$:= **true**
▮ **skip** -> $done_{1,a}$:= **false**]

$c_{1,b}$ = [P_a!*pebble*() -> $done_{1,b}$:= **true**
▮ **skip** -> $done_{1,b}$:= **false**]

$c_{2,a}$ = [P_b?*pebble*() -> $done_{2,a}$:= **true**
▮ **skip** -> $done_{2,a}$:= **false**]

$c_{2,b}$ = [P_a!*pebble*() -> $done_{2,b}$:= **true**
▮ **skip** -> $done_{1,b}$:= **false**]

P_1 = $[P_a::c_{1,a} \parallel P_b::c_{1,b}]$

P_2 = $[P_a::c_{2,a} \parallel P_b::c_{2,b}]$

P = $[\ P_a::[c_{1,a}\ ;\ c_{2,a}] \parallel P_b::[c_{1,b}\ ;\ c_{2,b}]]$

Dans chaque système P_i, $i = 1, 2$, les variables $done_{i,a}$ et $done_{i,b}$ ont la même valeur à la terminaison. Par contre, P peut terminer dans un état où $done_{1,a}$ vaut *true*, $done_{1,b}$ vaut *false*, $done_{2,a}$ vaut *true* et $done_{2,b}$ vaut *false*. L'état correspondant alors à la terminaison de P_1 est caractérisé par $done_{1,a}$ valant *true* et $done_{1,b}$ valant *false*, qui n'est pas un état de P_1!

Remarquons aussi que la considération de la convention de terminaison répartie conduirait ici à des problèmes sémantiques difficilement surmontables. Reprenons l'exemple précédent avec

$c_{1,a}$ = *[P_b?*pebble*() -> **skip**]

$c_{1,b}$ = **skip**

$c_{2,a}$ = **skip**

$c_{2,b}$ = *[P_a?*pebble*() -> **skip**]

Alors P_1 et P_2 terminent proprement, alors que P se bloque! En effet, la terminaison de $c_{1,b}$ qui forçait celle de $c_{1,a}$ s'est transformée en une simple transmission du contrôle à $c_{2,b}$. Cette remarque est donc un argument supplémentaire pour écarter de ce travail la convention de terminaison répartie.

5.1.2. Séquencement réparti

Nous pouvons donc poser la définition suivante inspirée de Elrad et Francez ([EF 82]). Des définitions analogues ont été proposées indépendamment de notre travail par Gerth et Shrira ([GS 86]).

Définition 5.1.2.1. *Soit G un réseau et*

$$P_i = [P_a::c_{i,a} \parallel P_b::c_{i,b} \parallel \ldots]$$

pour $i = 1, 2$, deux systèmes répartis construits sur G. La composition séquentielle (répartie) de P_1 et P_2 est le système P défini par

$$P = [\ P_a :: \ [c_{1,a}\ ;\ c_{2,a}] \parallel P_b :: \ [c_{1,b}\ ;\ c_{2,b}] \parallel \ldots]$$

Cette définition suppose bien sûr que P est un système syntaxiquement correct. Par construction, P est construit lui aussi sur G. Dans la pratique on notera $P_{i,a}$ le processus associé au noeud a de G dans le système P_i. On notera $P = P_1;;P_2$.

Nous formalisons maintenant la discussion de la section 5.1.1.

Définition 5.1.2.2. *Soit $P = P_1;;P_2$. P est dit synchronisé si, dans tout calcul de P, aucun message n'est échangé entre deux processus $P_{1,u}$ et $P_{2,v}$.*

La propriété de synchronisation est bien sûr indécidable en général, puisqu'elle est équivalente à l'arrêt d'un certain programme. Comme la symétrie, c'est une propriété sémantique. Il nous faut donc définir des conditions suffisantes la garantissant. Les méthodes de Apt ([Apt 83]) fondées sur l'analyse des chemins du système partiellement interprété seraient sans doute applicables ici. Nous préférons une condition plus simple, suffisante pour la pratique .

Définition 5.1.2.3. *Soit $P = P_1;;P_2$. P_1 et P_2 sont dits disjoints si aucun nom de canal n'apparaît à la fois dans P_1 et dans P_2.*

Il faut noter que, dans la sémantique de CSP présentée en section 1.3, les noms des canaux *chan* jouent le rôle de variables muettes. Un renommage ne change pas les calculs d'un système. A un renommage près, on peut donc toujours supposer que deux systèmes donnés sont disjoints.

Propriété 5.1.2.1. *Soit $P = P_1;;P_2$. Si P_1 et P_2 sont disjoints, alors P est synchronisé.*

Cette condition n'est évidemment pas nécessaire comme le montre l'exemple suivant.

$P_{1,a}$ :: $[P_b!chan()]$
$P_{1,b}$ :: $[P_a?chan()]$
$P_{2,a}$ :: $[P_b!chan()]$
$P_{2,b}$ :: $[P_a?chan()]$

P_1 :: $[P_{1,a} \parallel P_{1,b}]$
P_2 :: $[P_{2,a} \parallel P_{2,b}]$
P = $P_1;;P_1$

La propriété fondamentale ci-dessous exprime qu'un système synchronisé réalise en fait exactement une succession de phases synchrones ("lock-step phases").

Propriété 5.1.2.2. *Soit $P = P_1;;P_2$ un système synchronisé. Alors tous calcul fini de P est causalement équivalent à la concaténation d'un calcul de P_1 et d'un calcul (d'un sous-système) de P_2.*

Preuve Soit λ une transition de P. Si λ est de la forme $\{P_u :: \mu\}$ et si μ est une commande de $P_{i,u}$, alors nous dirons que λ est une transition de P_i. Si λ est de la forme $\{P_u :: \mu_u, P_v :: \mu_v\}$, alors c'est une communication entre P_u et P_v. Puisque P est synchronisé, μ_u et μ_v sont des transitions du même système P_i, et nous dirons que λ est une transution de P_i.

Soit C un calcul fini de P, λ_2, λ_1 deux transitions successives de C. Supposons que λ_2 soit une transition de P_2, et λ_1 de P_1. Montrons qu'elles sont nécessairement concurrentes. Si P_u est actif dans λ_2, cela signifie que le contrôle est déjà passé de $P_{1,u}$ à $P_{2,u}$ dans P_u. Donc λ_1 ne peut inclure une transition de $P_{1,u}$, et P_u est passif dans λ_1. Donc λ_2 et λ_1 commutent et l'on conclut par finitude de C. ∎

C se met donc sous la forme $C_1;C_2$ où C_1 est un calcul de P_1. C_2 est un calcul de P où seuls sont actifs les processeurs P_u qui terminent dans C_1. C_2 peut être vu comme un calcul du sous-système de P_2 formé des seuls processeurs qui terminent dans C_1. Si tous les calculs de P_1 et P_2 terminent proprement, alors il en est de même pour P et tout calcul maximal de P est causalement équivalent à une concaténation $C_1;C_2$ de calculs maximaux de P_1 et P_2, l'état initial de C_2 étant l'état final de C_1.

Le système P_1 peut donc être vu comme un "frontal" pour P_2, chargé de préparer l'état initial de P_2. Remarquons aussi que par les théorèmes de la section 2.4, il est imposible pour les processus de discerner si les calculs C_1 et C_2 se sont effectivement succédés, ou se sont interpénétrés. Du point de vue des processus, $P_1;;P_2$ est donc indiscernable dans ce cas de la composition séquentielle centralisée des deux phases P_1 ; P_2. Ceci n'est pas vrai si par exemple P_1 peut se bloquer dans un état où certains de ses processus ont cependant terminé.

Propriété 5.1.2.3. *Soit $P = P_1;;P_2$ un système synchronisé. Supposons que tous les calculs maximaux de P_1 et de P_2 sont finis et terminent proprement. Alors tous les calculs maximaux de P sont finis, terminent proprement, et sont causalements équivalents à une concaténation $C_1; C_2$ où C_1 est un calcul terminé de P_1 et C_2 un calcul terminé de P_2, d'état initial l'état final de C_1.*

5.1.3. Itération répartie

Exactement comme nous avons déduit la composition séquentielle répartie de la notion centralisée classique, nous pouvons considérer l'itération répartie d'un système. Cette construction correspond typiquement à un comportement oscillant d'un système, où les vagues de changement de phase se déplacent à la surface du réseau. Nous rencontrerons de nombreuses applications de cette construction aux chapitre 6, 7 et 8.

Définition 5.1.3.1. *Soit*

$$P = [P_a::c_a \parallel P_b::c_b \parallel \ldots]$$

un système réparti construit sur un réseau G. Soit B un vecteur d'expressions booléennes $\langle b_a, b_b, \ldots\rangle$ L'itération répartie de P par rapport à B est le système

$$Q = [\ P_a :: \ *[b_a \texttt{ -> } c_a] \parallel P_b :: \ *[b_b \texttt{ -> } c_b] \parallel \ldots]$$

Ce nouveau système sera noté $**[B \texttt{ -> } P]$.

Cette proposition suppose bien sûr qu'il soit syntaxiquement correct. En pratique, chaque b_u sera construit sur les variables de c_u, ce qui garantit la correction. Le problème de la synchronisation est ici beaucoup plus complexe que précédemment. Nous dirons qu'un processus $*[b_u \texttt{ -> } P_u]$ est dans sa phase p s'il a déjà itéré $(p-1)$ fois sa boucle externe.

Définition 5.1.3.2. *Une itération répartie est dite synchronisée si aucune communication n'a lieu entre des processus en phases différentes.*

Considérons par exemple le système $P = [P_a \parallel P_b]$ avec

P_a :: [P_b!*chan*() -> **skip**
▮ P_b!*chan*() -> P_b!*chan*()]

P_b :: [P_a?*chan*() -> **skip**
▮ P_a?*chan*() -> P_a?*chan*()]

et $B = \langle true, true\rangle$. L'itération répartie $**[B\texttt{->}P]$ n'est pas synchronisée. En effet, P_a peut envoyer à P_b un second message durant sa première phase, qui sera reçu par P_b dans sa seconde phase. Par contre, si nous différencions les deux possibilités, nous garantissons la synchronisation.

P_a :: [P_b!$chan_0$() -> **skip**
▮ P_b!$chan_1$() -> P_b!$chan_1$()]

P_b :: [P_a?$chan_0$() -> **skip**
▮ P_a?$chan_1$() -> P_a?$chan_1$()]

Des conditions suffisantes de synchronisation sont beaucoup plus complexes à obtenir que dans le cas précédent. Nous étudions tout d'abord les propriétés des systèmes synchronisés. La propriété ci-dessous est l'analogue de la propriété 5.1.2.2.

Propriété 5.1.3.1. *Soit Q=$**[B->P]$ une itération répartie synchronisée. Tout calcul fini de Q est causalement équivalent à une concaténation (de calculs de sous-systèmes) de P.*

Preuve Comme pour la propriété 5.1.2.2, on associe, grâce à la condition de synchronisation, une phase à chaque transition. On montre ensuite que deux transitions λ_2 et λ_1 associées à des phases p_2 et p_1 avec $p_1 < p_2$, sont nécessairement concurrentes, et peuvent donc être commutées. Les seules processus actifs en phase p sont ceux qui ont atteint leur $p-1$ premières phases. ∎

Nous voulons maintenant relier les calculs maximaux de Q = **[B->P] à ceux de P, par une propriété analogue à la propriété 5.1.2.3. Il faut évidemment que les conditions booléennes de B se comportent elles aussi de manière synchronisée.

Définition 5.1.3.3. *Soit Q = **[B->P] une itération répartie synchronisée. On dit que B est synchronisée dans Q si, au début de chaque phase, la condition b_u est vraie pour tous les processus P_u, ou fausse pour tous les processus P_u.*

Propriété 5.1.3.2. *Soit Q = **[B->P] une itération répartie synchronisée, où B est synchronisée. Supposons que tous les calculs maximaux de P soient finis et terminent proprement, et que tous les calculs de Q soient finis. Alors les calculs maximaux de Q sont causalement équivalents à une concaténation*

$$C_1; C_2; \ldots; C_p$$

de calculs maximaux terminés de P, où l'état initial de C_{i+1} est l'état final de C_i.

Il nous reste maintenant à déterminer des conditions suffisantes de synchronisation. Nous nous inspirons des conditions proposées par [GS 86] ("general tail-closeness"). Le problème est de permettre à un processus P_u de déterminer dans quelle phase est son voisin P_v, uniquement sur la base des communications entre P_u et P_v.

Définition 5.1.3.4. *Soit P un système réparti. un calcul C est dit équivoque si, dans l'état final de C, un processus P_u est terminé tandis qu'un autre processus P_v est prêt à communiquer avec P_u.*

Propriété 5.1.3.2. *Si P n'a aucun calcul équivoque, alors son itération répartie Q = **[B->P] est synchronisée.*

Preuve Soit C un calcul de Q violant la condition de synchronisation. On peut supposer C fini. Supposons que la première communication de processus en phases distinctes soit entre P_u et P_v, respectivement en phases p et q, $p < q$. Restreignons C aux transitions antérieures à cette communication. Nous obtenons un calcul synchronisé. Par la propriété 5.1.3.1, il est équivalent à une concaténation de calculs $C_1; C_2; \ldots$ dans l'état final de C_p, P_u est prêt à communiquer avec P_v, alors que P_v est prêt à passer en phase $p+1$. ∎

Remarquons que cette propriété n'est pas nécessaire, comme le montre l'exemple suivant.

P = [P_a ‖ P_b]

```
Pa :: [ Pb!chan1()
        [ Pb!chan2() -> skip
        ▮ true -> skip ]]

Pb :: [ Pa?chan1()
        [ Pa?chan2() -> skip
        ▮ true -> skip ]]
```

Le problème est donc de déterminer des conditions suffisantes qui garantissent qu'un système n'a pas de calcul équivoque. Soit S (pour "synchronisateur") le système présenté en figure 5.1.3. Il est facile de vérifier qu'il n'a pas de calcul équivoque. Un tel système peut être utilisé pour rendre non équivoque un autre système.

Propriété 5.1.3.3. *Soient P_1 et P_2 deux systèmes disjoints construits sur un même réseau G. Suppsons que dans tout calcul maximal de P_2, au moins un message est échangé sur chaque canal de G. Si P_2 n'a aucun calcul équivoque, alors leur composition séquentielle répartie $P_1;;P_2$ non plus.*

Preuve Soit C un calcul fini équivoque de $P_1;;P_2$. dans l'état final de C, P_u est terminé et P_v est prêt à communiquer avec P_u. Par la propriété 5.1.2.2, C est équivalent à une concaténation $C_1; C_2$ d'un calcul de P_1 et d'un calcul de P_2. Comme P_u a terminé, $P_{2,u}$ a été exécuté, et donc au moins un message a été echangé avec $P_{2,v}$. Donc le calcul C_2 est équivoque. ∎

Nous avons déjà mentionné que la condition de disjonction est purement syntaxique. Tout système peut donc être transformé en un système n'ayant aucun calcul équivoque, au prix de l'ajout d'un phase supplémentaire de synchronisation.

```
S_u ::  [ received:= false; sent:= false;
          *[ ▌ in  ¬received[in]; S_in?chan() -> received[in]:= true
           ▌ out ¬sent[out]; S_out!chan() -> sent[out]:= true
          ]]
```

Figure 5.1.3.

5.2. Propriétés

Nous montrons dans cette section que les deux opérations fondamentales définies ci-dessus préservent la modularité et la symétrie.

5.2.1. Conservation de la modularité

Dans ce qui suit on fixe une famille $\mathcal{G}$ de réseaux. Soit $\mathcal{P}_1$ et $\mathcal{P}_2$ deux familles de système batis sur des réseaux de $\mathcal{G}$. On note $\mathcal{P}_1$;; $\mathcal{P}_2$ la famille obtenue en formant, pour chaque réseau G de $\mathcal{G}$, la composition séquentielle répartie des systèmes P_1 et P_2 bâtis sur G dans $\mathcal{P}_1$ et $\mathcal{P}_2$.

Propriété 5.2.1.1. *Si $\mathcal{P}_1$ et $\mathcal{P}_2$ sont modulaires, alors leur séquencement réparti $\mathcal{P}_1$;; $\mathcal{P}_2$ l'est aussi.*

Soit $\mathcal{P}$ une famille de systèmes batis sur $\mathcal{G}$. Supposons donné, pour chaque système P de $\mathcal{P}$, un vecteur de conditions boléennes B, une pour chaque processus de P. Soit $\mathcal{B}$ la famille ainsi formée. On dit que $\mathcal{B}$ est **modulaire** si, pour tous réseaux G, G' de $\mathcal{G}$, et pour tous systèmes P, P' de $\mathcal{P}$ bâtis respectivement sur G et G', les conditions booléennes b_u et b'_u associés aux processus P_u et P_u sont syntaxiquement identiques pour tout sommet u de $G \cap G'$. On note $**[\mathcal{B}$ -> $\mathcal{P}]$ la famille de systèmes obtenus en formant, pour tout réseau G de $\mathcal{G}$, l'itération répartie du système P de $\mathcal{P}$ bâti sur G par rapport au vecteur associé bb de $\mathcal{B}$.

Propriété 5.2.1.2. *Si $\mathcal{P}$ est modulaire et $\mathcal{B}$ est modulaire, alors $**[\mathcal{B}$ -> $\mathcal{P}]$ l'est aussi.*

On remarquera de plus que la maquette présentée en figure 5.1.3. définit en fait une famille modulaire de système. Par la propriété 5.2.1.1, il est donc possible transformer chaque système d'une famille modulaire en un système non équivoque tout en préservant la modularité de cette famille (ceci suppose bien sûr que l'ensemble *Name* ne puisse être totalement utilisé par les systèmes de cette famille).

5.2.2. Conservation de la symétrie

Soit G un réseau fixé, et soit P_1 et P_2 des systèmes construits sur G. Nous voulons relier la symétrie de la composition séquentielle répartie $P = P_1;;P_2$ avec celle de P_1 et de P_2. Dans un calcul de P, l'état initial de P_2 dépend de l'état final de P_1. Pour garantir le transport des calculs partiels par une action S_σ, il nous faut donc exprimer que l'état final de P_1 est lui aussi transporté correctement par S_σ. Ceci nous conduit donc à introduire une notion de symétrie *fonctionnelle* qui exprime non seulement la symétrie des calculs, mais aussi celle des états initiaux et finaux.

Définition 5.2.2.1. *Soit P un système réparti construit sur un réseau G. P est dit Σ-fonctionnellement symétrique si pour tout $\sigma \in \Sigma$, il existe des bijection S_σ sur $PCOMP$ et E_σ sur Env telles que les propriétés 1, 2, 3 et 4 de la définition 2.3.1 soient satisfaites et de plus*

5) Si C est un calcul maximal d'environement initial env_0 et (si C est fini) d'environnement final env_1, alors $S_\sigma(C)$ est un calcul d'environnement initial $E_\sigma(env_0)$ et final $E_\sigma(env_1)$.

Dans les systèmes symétriques que nous étudierons aux chapitres 6 et 7, on peut toujours supposer que les variables de travail reçoivent des valeurs arbitraires à la terminaison. Seules les variables *data* au chapitre 6 et *decision* au chapitre 7 sont significatives pour les problèmes qui nous intéressent. Sous cette transformation, ces algorithmes symétriques sont en fait fonctionellement symétriques. L'action E est triviale au chapitre 6, et, au chapitre 7, E_σ change la valeur u d'une variable *decision* en $\sigma(u)$. De fait, nous

confondrons en général la symétrie fonctionnelle et la symétrie simple. On remarquera que les conditions suffisantes de la section 4.4.3 garantissent en fait la symétrie fonctionnelle. En d'autres termes, dans la pratique, les algorithmes symétriques sont en fait généralement fonctionnellement symétriques. Dans ce qui suit, on suppose fixée une action E de Σ sur Env.

Propriété 5.2.2.1. *Soit P_1 et P_2 des systèmes Σ-fonctionnellement symétriques dont tous les calculs terminent proprement, et $P{=}P_1;;P_2$. Si P est synchonisé, alors il est aussi Σ-fonctionnellement symétrique.*

Preuve Soit C un calcul de P de la forme $C_1;C_2$, où C_1 et C_2 sont des calculs de P_1 et (d'un sous-système) de P_2. Soit *env* l'etat final de C_1. Posons

$$S_\sigma(C) = S_\sigma(C_1); S_\sigma(C_2)$$

Cette concanénation est bien définie. En effet, $P_{1,u}$ termine dans C_1 si et seulement si $P_{1,\sigma(u)}$ termine dans $S_\sigma(C_1)$. $P_{2,u}$ est actif dans dans la transition k de C_2 si et seulement si $P_{2,\sigma(u)}$ l'est dans la transition k de $S_\sigma(C_2)$. *env* est aussi l'état initial de C_2. $E_\sigma(env)$ est donc, à la fois, l'état final de $S_\sigma(C_1)$ et l'état initial de $S_\sigma(C_2)$.

Soit maintenant C un calcul quelconque de P. Par la propriété 5.1.2.2, il existe une permutation telle que C se mette sous la forme $C' = C_1; C_2$ ci-dessus. Définissons $S_\sigma(C)$ comme l'image par la permutation inverse de $S_\sigma(C')$. En remarquant que P_u est actif dans la transition k de C' si et seulement si $P_{\sigma(u)}$ l'est dans la transition k de $S_\sigma(C')$, on vérifie que $S_\sigma(C)$ est bien défini.

On vérifie ensuite les propriétés 2, 3 et 4 de la définition 4.3.1 sans difficulté. Enfin, l'état initial de C est celui de C_1 et son état final celui de C_2. La propriété 5 la définition 5.2.2.1 en découle facilement. ∎

Une méthode analogue, en utilisant cette fois la propriété 5.1.3.2, conduit au résultat suivant. Soit P un système réparti et B un vecteur de conditions booléennes pour P. On dit que B est Σ-symétrique si B_u est satisfaite dans un environnement *env* si et seulement si $B_{\sigma(u)}$ est satisfaite dans $E_\sigma(env)$ pour tout $\sigma \in \Sigma$.

Propriété 5.2.2.2. *Soit P un système Σ-fonctionnellement symétrique dont tous les calculs terminent proprement et B un vecteur Σ-symétrique pour P. Soit Q=**[B -> P] l'itération répartie de P. Si Q est synchonisé et si tous ses calculs sont finis alors Q est Σ-fonctionnellement symétrique.*

En fait, la condition de finitude sur Q n'est due qu'aux restrictions du théorème 2.3.4.1. Nous pensons que cette propriété reste vraie sans cette condition, mais nous ne sommes pas en mesure de le prouver dans le cadre de ce travail.

5.3. Applications

5.3.1. Modularisation

Nous pouvons utiliser la composition séquentielle répartie pour ajouter à un algorithme donné une phase préparatoire, permettant ainsi aux processus de disposer d'un certain nombre d'informations à l'initialisation de l'algorithme proprement dit. Ces informations peuvent, en particulier, concerner le réseau dans lequel les processeurs sont plongés. Les résultats de la section 6.5 montrent qu'il existe un algorithme $\mathcal{E}$ modulaire et (fonctionnellement) symétrique en $CSP_{i/o}$ pour les réseaux fortement connexes tel que

1) tous les calculs se terminent proprement;
2) à la terminaison, chaque processus possède dans sa variable *graph* une image du réseau sous-jacent.

Un tel algorithme est appelé algorithme d' **apprentissage**. On peut par exemple supposer que la variable *graph* est un enregistrement de la forme mentionnée en section 2.1.

Considérons maintenant une famille $\mathcal{P}$ de systèmes non modulaire. L'absence de modularité exprime que les processus des systèmes de $\mathcal{P}$ utilisent des informations concernant le réseau dans lequel ils sont plongés. Ces informations peuvent être de deux types. Elles sont **implicites** si elles sont en fait codées dans les structures de contrôle. Par exemple, de nombreuses solutions au problème de la détection de la terminaison repartie utilisent le fait que les processus sont nommés de telle sorte qu'ils forment un circuit hamiltomien du réseau (*cf.* [AR 84] par exemple). Un procesus P_i reçoit donc les vagues de contrôle de P_{i-1} et les transmet à P_{i+1}. L'existence d'un circuit hamiltonien est donc implicite dans la structure même de l'algorithme.

Par contre, d'autres algorithmes utilisent cette information par le biais de la valeur d'une variable, supposée prédéfinie lors de l'initialisation des systèmes. C'est le cas de la plupart des algorithmes qui utilisent la taille N du réseau (*cf.* [Bougé 86]). On dit alors que l'information est **explicite**. Soit P un système bâti sur un réseau G. Supposons que chaque processus P_u de P declare une variable $graph_u$ initialisée avec précisement la valeur G. On dit que P utilise explicitement G et on note $P(G)$. $P(.)$ désigne le système obtenu en détruisant cette initialisation (c'est en fait un schéma de système).

Définition 5.3.1. *Soit $\mathcal{G}$ une famille de réseaux et $\mathcal{P}$ une famille de systèmes construits sur les réseaux de $\mathcal{G}$ utilisant explicitement leur réseau sous-jacent. On dira que $\mathcal{P}$ est une famille explicitement non modulaire.*

Nous avons défini en section 2.2 le résultat d'un système dans un calcul proprement terminé. Soit P et Q deux systèmes construits sur un même réseau G dont tous les calculs terminent proprement. Soit *Res* un ensemble de variables. Nous dirons que P et Q sont **équivalents** (modulo *Res*) si l'ensemble des résultats de P et de Q restreints aux variables de *Res* sont les mêmes. Nous pouvons maintenant énoncer le théorème de modularisation.

Théorème 5.3.1.1. *Soit $\mathcal{G}$ une famille de réseaux fortement connexes. Soit $\mathcal{P}$ une famille explicitement non modulaire de systèmes en $CSP_{i/o}$ dont tous les calculs terminent. Pour tout système $P(G)$ de $\mathcal{P}$, il existe un système Q équivalent à P par rapport aux variables de P tel que la famille $\mathcal{Q}$ ainsi formée soit modulaire. De plus, si $\mathcal{P}$ est symétrique alors $\mathcal{Q}$ l'est aussi.*

Preuve Soit $P(G)$ un système explicitement non modulaire construit sur G. Soit E le système associé au réseau G dans l'algorithme d'apprentissage $\mathcal{E}$. Il suffit de prendre $Q = E;;P(.)$. La valeur de G utilisée par $P(.)$ est celle produite comme résultat par E. ∎

Remarquons que l'équivalence est en fait beaucoup plus forte puisque tout calcul de Q est (causalement équivalent à) un calcul de P préfixé par un calcul de E. Il y a donc équivalence au niveau *abstrait* des systèmes non interprétés. Le problème est que nous ne savons pas exprimer simplement de telles équivalences (on rencontre une situation similaire dans [OA 86]). Ce théorème montre qu'en $CSP_{i/o}$, il est possible de remplacer l'utilisation d'information globales "innées" par un apprentissage explicite de ces informations. Il montre aussi que le problème de la diffusion, étudié au chapitre 6, capture l'essence de la notion de modularité.

5.3.2. Symétrisation

Le problème de la symétrisation est plus difficile que celui de la modularisation. L'absence de symétrie peut être due en général au fait qu'un *processeur* jouit d'un certain privilège dans les calculs d'un système. Plus exactement, le *processus* associé à ce processeur présente un comportement privilégié. L'absence de symétrie est donc due au lien entre processeur et processus, qui, en CSP, est statique. Le processus P_u est statiquement associé au processeur u. Si, par contre, ce lien était dynamique, ce processus pourrait migrer sur l'un quelconque des processeurs. Aucun de ceux-ci ne serait privilégié, puisque tous pourraient jouer le rôle initialement dévolu au processeur u. La méthode de symétrisation que nous présentons est fondée sur ce principe.

Soit G un réseau, et

$$P :: \quad [P_a::c_a \parallel P_b::c_b \parallel \ldots]$$

un système construit sur G. Soit σ un automorphisme de G. Soit $c_{\sigma,u}$ la commande obtenue en transformant dans c_u chaque commande de communication $P_u?message$ en $P_{\sigma(u)}?message$ et $P_u!message$ en $P_{\sigma(u)}!message$. Soit P_σ le système réparti suivant.

$$P_\sigma :: \quad [\ P_{\sigma(a)}::c_a \parallel P_{\sigma(b)}::c_b \parallel \ldots]$$

remarquer que les conditions de correction syntaxique sont satisfaites par P_σ, et que P_σ est un système réparti bâti sur G. Au renommage de P_u en $P_{\sigma(u)}$ près, il est clair que P et P_σ ont les mêmes calculs. Seule l'association entre processus et processeurs a été modifiées. L'ancien processus P_u est maintenant exécuté par le processeur $\sigma(u)$. Il suffit maintenant de permettre aux processeurs de déterminer dynamiquement l'association choisie, de telle sorte que la négociation soit symétrique. Un moyen simple est par exemple de procéder à une élection symétrique, et de laisser le processeur élu prendre la décision et en informer les autres processeurs du réseau. Le chapitre 7 sera entièrement consacré au problème de l'élection symétrique.

Propriété 5.3.2.1. *(symétrisation par allocation dynamique) Soit G un réseau fortement connexe admettant un système électoral symétrique en $CSP_{i/o}$. Soit P un système réparti construit sur G en $CSP_{i/o}$ dont tous les calculs terminent. Alors il existe un système symétrique Q, équivalent à P par rapport aux variables de P.*

Preuve Le système Q est obtenu par composition séquentielle répartie de trois phases fonctionnellement symétriques et disjointes. Les seules variables communes de ces trois phases sont *decision*, qui contient le nom du processeur élu en phase 1, et *which*, qui contient l'automorphisme choisi par ce processeur.

Phase 1

Les processeurs exécutent une élection fonctionnellement symétrique (section 7.4.1). A la terminaison, chaque processeur P_u possède le nom du processeur élu dans sa variable $decision_u$. Par définition, si un calcul maximal C aboutit à l'élection de P_v, alors le calcul $S_\sigma(C)$ obtenu par symétrie aboutit à celle de $P_{\sigma(v)}$. En d'autres termes, E_σ change $decision_u$ en $\sigma(decision_u)$.

Phase 2

Le processeur P_u élu en phase 1 (tel que $decision_u = u$) choisit un automorphisme ρ de G de manière non déterministe, et le diffuse grâce à l'algorithme de diffusion symétrique en $CSP_{i/o}$ pour les réseaux fortement connexes de la section 6.2.1, dans lequel P_u joue le rôle de P_0. Chaque processeur possède alors cet automorphisme dans sa variable *which*. L'action S est la suivante. Soit C un calcul où $decision_u = v$ dans l'état initial, et où l'automorphisme ρ est choisi et diffusé par P_v. Alors, dans le calcul $S_\sigma(C)$, l'automorphisme $\sigma.\rho$ est choisi et diffusé par $P_{\sigma(v)}$. Donc E_σ change $which_w$ en $\sigma(which_w)$.

Phase 3

Chaque processeur $P_{\rho(u)}$ exécute le processus $c_{\rho,u}$, où ρ est la valeur de $which_u$. L'action S est la suivante. Si C est un calcul où *which* vaut ρ, laors $S_\sigma(C)$ est le même calcul où initialement *which* vaut $\sigma.\rho$, et où chaque processeur $P_{\sigma.\rho(u)}$ se comporte exactement comme le processeur $P_{\rho(u)}$ dans C, et exécute $c_{\sigma.\rho,u}$.

Par la propriété 5.1.2.2, les calculs maximaux de Q sont causalement équivalents à une concaténation de calculs maximaux de chacunes de ces phases. Tous les calculs de P peuvent être effectués en phase 3. Donc, restreints aux variables de P, les états finaux de Q sont exactement ceux de P. ∎

Il est important de remarquer que cette procédure de symétrisation ne préserve pas la modularité des familles de systèmes. De plus, les systèmes produits ne sont pas, en général, explicitement non modulaires, et la propriété 5.3.1.1 ne s'applique pas ici. En effet, Q_u contient syntaxiquement une alternative indexée par l'ensemble des automorphismes du réseau G. La description d'une procédure de symétrisation qui respecte la modularité reste un problème ouvert.

Les résultats précédents montrent que le problème de l'élection symétrique, étudié au chapitre 7, capture l'essence de la notion de symétrie. Nous verrons qu'une large classe de réseaux admet un système électoral symétrique en $CSP_{i/o}$: les cliques et les anneaux de longueur un nombre premier en particulier. Cependant, pour certains types d'algorithmes, il est possible de lever cette restriction. On dira qu'un système est **fonctionnel** si tous ses calculs terminent proprement, et ne produisent qu'un *unique* résultat (ont tous même état final). Les systèmes de diffusion présentés au chapitre 6 sont par exemple fonctionnels. Nous dirons qu'un systéme P est en forme normale si chaque processus P_u est de la forme

```
P_u :: [ Init
         *[ ▮_i Guard_i -> Command_i
       ]]
```

où toutes les communications sont dans les gardes $Guard_i$. On trouvera dans [AC 85] une transformation permettant de transformer tout système écrit en en un système équivalent en forme normale. Il n'est donc pas restrictif de se restreindre à ne considérer que de tels systèmes ici.

Propriété 5.3.2.2. *(symétrisation par superposition) Soit P un système réparti fonctionnel en forme normale. Alors il existe un système Q symétrique, équivalent à P par rapport aux variables de P.*

Preuve Soit Σ le groupe des automorphismes de G. Pour chaque automorphisme σ de G, nous produisons une copie $P_{\sigma,u}$ de P_u en indexant tous les noms de variables et de canaux de P_u avec σ.

$P_{\sigma,u}$ :: [*Init*
*[$\underset{i}{\blacksquare}$ $Guard_{\sigma,i}$ -> $Command_{\sigma,i}$
]]

Soit P_σ le système obtenu en composant les $P_{\sigma,u}$. Soit maintenant Q_u le processus suivant

Q_u ::[*INIT*
*[$\underset{\rho,i}{\blacksquare}$ $Guard_{\rho,i}$ -> $Command_{\rho,i}$
]]

où *INIT* est la composition séquentielle des commandes $Init_\rho$ dans un ordre arbitraire. Soit alors Q le système suivant.

Q :: $[Q_a \parallel Q_b \parallel \ldots]$

On peut montrer que tout calcul C de Q est causalement équivalent à une concaténation de calcul C_ρ de P_ρ, où ρ parcourt Σ.

Le système Q est symétrique. Soit C un calcul de Q. $S_\sigma(C)$ est le calcul où la copie $P_{\sigma.\rho,u}$ se comporte comme la copie $P_{\rho,u}$ dans C.

Enfin, en identifiant $P_{Id,u}$ et P_u, les calculs de Q contiennent exactement les calculs de P. ∎

La symétrisation par allocation différée brise la symétrie (de manière symétrique) avant-même le début de l'algorithme proprement dit. La symétrisation par superposition revient à exécuter en parallèle les algorithmes obtenus en considérant *toutes* les allocations possibles. Ces deux méthodes se placent donc aux extrémités de l'éventail des possibilités, et n'ont sans doute que peu d'intérêt directement pratique. Elles peuvent cependant servir de base à d'autres optimisations.

En fait, ce qui est utile en pratique est une forme de symétrisation par *coopération*. Au départ, toutes les copies existent (virtuellement au moins) comme dans la symétrisation par superposition. Les copies sont alors dynamiquement éliminées ou fusionnées, jusqu'à ce qu'une seule d'entre elles demeure, comme dans la symétrisation par allocation différée. Tout le problème est de gérer l'élimination ou la fusion symétriquement. On trouvera une application de cette approche dans le travail de Tan et van Leeuwen ([TL 86]). Malheureusement, le mécanisme d'élimination employé n'est pas symétrique dans notre sens. Par contre, l'algorithme de Shavit et Francez ([SF 86]) est un exemple très élégant de symétrisation par coopération, à partir de l'algorithme de [DS 80]. La définition exacte d'une telle méthode de symétrisation reste encore un problème ouvert.

6. Algorithmes modulaires de diffusion

Nous avons défini au chapitre 3 la notion de modularité dans le cadre des systèmes répartis en CSP. Nous en avons donné une définition syntaxique: une famille de systèmes (un algorithme) est modulaire si la forme *syntaxique* des processus composant les systèmes de la famille ne dépend que de l'environnement local de ceux-ci dans le réseau. En d'autres termes, l'algorithme général défini par cette famille n'utilise aucune connaissance initiale concernant le graphe de communication dans sa globalité.

Le but de chapitre est d'évaluer cette notion à propos d'un problème classique, celui de la diffusion. Il s'agit pour un processeur donné du réseau, appelé *initiateur*, de transmettre une donnée à tous les autres processeurs du réseaux. Ce problème admet évidemment une solution triviale si aucune condition n'est imposée. La modularité exprime que le processeur initiateur ne peut utiliser aucune information concernant la topologie globale du réseau pour guider sa diffusion. En un certain sens, il doit donc travailler "à l'aveuglette". Nous examinerons de plus la symétrie des solutions obtenues. Tous les processeurs, autres que l'initiateur, bien sûr, doivent alors jouer des rôles équivalents dans la diffusion.

Nous nous intéresserons aux trois dialectes de CSP que nous avons déjà mentionnés:
— CSP_{no} où aucune garde de communication n'est admise;
— CSP_{in} où seules les gardes de réception sont admises, mais non celles d'émission;
— $CSP_{i/o}$ où les deux types de gardes sont admis.

Nous définissons tout d'abord la notion d'algorithme de diffusion modulaire et symétrique. Nous montrons que les réseaux non connexes n'admettent pas d'algorithme de diffusion. Contrairement à ce que l'on pourrait penser, les réseaux seulement faiblement connexes en admettent sous certaines conditions.

Nous étudions ensuite le cas de $CSP_{i/o}$. Dans ce cas, on peut trouver un algorithme de diffusion modulaire et symétrique pour les réseaux fortement connexes. Cette solution est fondée sur la stratégie dite du "front d'onde". Cet algorithme est à la base de nombreuses autres variantes. En particulier, on en déduit un algorithme de diffusion symétrique (mais non modulaire) pour les réseaux faiblement connexes. on montre aussi que l'on peut diffuser des suites finies ou infinies de données.

Le cas de CSP_{in} et de CSP_{no} est plus complexe. Nous décrivons un algorithme modulaire et symétrique en CSP_{in} pour les réseaux bidirectionnels connexes. Nous décrivons aussi une solution modulaire pour les réseaux fortement connexes en CSP_{no}. Il n'existe pas dans ce cas de solution symétrique.

Nous montrons ensuite que le problème peut être étendu au problème de la multi-diffusion, où chaque processeur diffuse sa propre donnée dans le réseau. Il existe une solution modulaire et symétrique en $CSP_{i/o}$ pour les graphes fortement connexes. Il n'existe pas de telle solution en CSP_{in}. Ces solutions sont obtenues sous réserve de l'existence d'un prédicat local permettent de détecter la terminaison. Un cas fondamental est celui où la valeur diffusée par chaque processeur est précisément son environnement local. Nous montrons que, dans ce cas, un tel prédicat peut être trouvé, ce qui correspond à la possibilité pour les proceseurs d'apprendre dynamiquement le réseau dans lequel ils sont plongés.

6.1. Le probleme de la diffusion

La notion de diffusion ("broadcast") est l'un des principaux paradigmes dans le domaine de l'algorithmique répartie. L'un des principaux cadres d'étude de ce problème est sans doute celui des réseaux non fiables, où les messages peuvent être endommagés, perdus, melangés *etc.* On parle en général plutôt de protocole de diffusion. Le problème est de garantir qu'un message émis par l'initiateur est accepté par tous les processeurs ou par aucun d'entre eux, et que les messages successivement acceptés le sont dans le même ordre. L'exemple le plus classique est sans aucun doute celui du "bit alterné" où deux processeurs échangent messages et accusés de réception sur une ligne non fiable (*cf.* [CES 83] pour une solution dans le cadre de CSP).

Une autre approche analogue à celle-ci est l'approche "byzantine". Plutôt que de considérer que les lignes de transmission peuvent ne pas être fiables, on considère que les processeurs eux-mêmes peuvent exhiber

des comportements aberrants. Le problème dit des "généraux byzantins" est précisément la diffusion par un initiateur (le "général") d'une donnée ([LSP 82]). La difficulté est que l'initiateur lui-même peut avoir un comportement aberrant, ce qui ne doit pas empêcher les autres processeurs, au moins ceux parmi eux qui sont corrects, de se mettre d'accord sur une donnée fictive. Les techniques employées dans ce type de problème sont essentiellement des techniques d'écho. Une décision est prise seulement si l'on sait que des processeurs en nombre suffisant sont prêts à prendre cette décision (*cf.* [TPS 84] pour un traitement très élégant de cette idée).

Un autre domaine d'application des protocoles de diffusion est celui des bases de données réparties. Une mise à jour souhaitée par l'un des sites doit se faire de manière consistante sur tous les sites ou sur aucun d'entre eux. Des protocoles de diffusion à deux phases sont utilisés. Dans un premier temps, l'initiateur diffuse "expérimentalement" la mise à jour pour demander leur avis aux sites concernés ("ballon d'essai" en politique). Si cet avis, reçu en retour, est favorable, une confirmation est diffusée. Sinon, une annulation est diffusée. On mesure ici l'importance de protocoles efficaces, vu leur fréquence d'utilisation!

En fait, la notion même de diffusion est prise comme primitive dans certains langages de programmation répartie, de préférence à celle de communication point-à-point considérée dans CCS et *CSP* par exemple. Le langage TCSP permet de définir un opérateur de multi-synchronisation qui correspond en définitive à la diffusion atomique d'un signal à tous les processus concernés. Le langage Esterel ([BC 84]) ne connaît qu'une forme de communication, qui est la diffusion cumulative et évanescente de signaux. Les réseaux locaux de type Ethernet utilisent la diffusion comme concept de base. La communication point-à-point n'est alors qu'une forme spéciale de diffusion où la donnée diffusée par le destinateur est ignorée par tous les processeurs sauf le destinataire ([SDRC 83]).

Le problème de la diffusion est sous-jacent à de nombreux autres problèmes classiques. Nous étudierons en section 6.4 le problème de l'apprentissage. Dans ce problème, chaque processeur doit apprendre dynamiquement le réseau dans lequel il est plongé. Une version légèrement différente de ce problème est traitée dans [WS 83]. Le problème du routage peut aussi être vu comme un problème de diffusion. Santoro et Khatib ([SK 82]) en proposent une solution où

a processor has no explicit knowledge of the location of other processors (except the neighbours)

et qui est pourtant optimale dans de nombreux cas. Le problème de la détermination d'un arbre recouvrant est lui aussi équivalent au problème de l'apprentissage. Enfin, la connaissance d'un algorithme d'apprentissage conduit trivialement à la solution de nombreux problèmes classiques tels que la recherche d'extrema ([BL 85]), la recherche de centre et de medians, et, de manière plus génerale, la reconnaissance de propriétés topologiques globales ([Angluin 80]).

A la lumière de cette discussion, nous pouvons maintenant donner une définition précise de la diffusion en *CSP*. Nous supposons que tous les réseaux que nous considérons dans ce cadre possèdent un sommet privilégié, noté conventionnellement 0, qui sera l'initiateur de la diffusion. Soit G un tel réseau. Nous noterons Σ_0 le groupe des automorphismes de G laissant son sommet 0 invariant. Nous supposerons aussi que, dans tous les systèmes répartis P que nous considèrerons, chaque processeur P_u déclare une variable $data_u$ destinée à contenir la donnée diffusée. Ces données sont des éléments d'un ensemble *Data* de cardinal au moins 2.

Définition 6.1.1. *Soit G un réseau et P un système réparti construit sur G. P est un algorithme de diffusion pour G si*

1) tous les calculs de P sont finis et terminent proprement;

2) si, dans l'état initial, $data_0 = d$, alors, dans l'état final, $data_u = d$ pour tous les processus P_u composant P.

On dira que P un algorithme de diffusion **symétrique** si de plus P est Σ_0-symétrique. Soit $\mathcal{G}$ une famille de réseaux et $\mathcal{P}$ une famille de systèmes répartis construits sur les réseaux de la famille $\mathcal{G}$. On dira que $\mathcal{P}$ est un algorithme (généralisé) de diffusion pour la famille $\mathcal{G}$ si chaque système P est un algorithme de diffusion pour le réseau G associé. On dira que cet algorithme est **modulaire** si la famille $\mathcal{P}$ est modulaire. On dira que cet algorithme est symétrique si chaque système P est symétrique.

Le problème de la diffusion que nous considérons peut s'énoncer ainsi.

Etant donné une famille de réseaux $\mathcal{G}$, existe-t-il un algorithme de diffusion modulaire, et si possible symétrique, pour $\mathcal{G}$?

Nous verrons que le réponse dépend crucialement du dialecte, $CSP_{i/o}$, CSP_{in} ou CSP_{no}, utilisé.

Avant de passer à l'étude détaillée des réponses à ce problème, nous établissons quelques résultats qui délimitent en quelque sorte le champ de l'étude. Le premier d'entre eux, intuitivement trivial, est qu'il n'est pas possible de diffuser une valeur dans un réseau non connexe. La méthode de preuve "par partition" employée est très proche de celle utilisée pour montrer l'impossibilité d'un accord dans un réseau non fiable où plus d'un tiers des processeurs peuvent avoir des comportements aberrants ([BenOr 83]).

Théorème 6.1.1. *Il n'existe pas d'algorithme de diffusion dans un réseau non connexe.*

Preuve Soit P un algorithme de diffusion pour un tel réseau G. Soit G_0 la composante (faiblement) connexe de P_0 dans G, et G_1 le reste de G (qui est non vide!). Deux proceseurs situés dans G_0 et G_1 ne peuvent échanger de message.

Soit C_0 et C_1 des calculs maximaux de P où P_0 diffuse respectivement des données d_0 et d_1. Remarquons que les transitions effectuées par les processeurs de G_0 et celles effectuées par ceux de G_1 sont nécessairement concurrentes, et donc commutent. Par commutation et restriction, on obtient à partir de C_0 un calcul C_0' où seuls les processeurs de G_0 sont actifs et terminent avec $data = d_0$. De même, à partir de C_1, on obtient un calcul C_1' où seuls les processeurs de G_1 sont actifs et terminent avec $data = d_1$. D'après la remarque ci-dessus, on peut fusionner C_0' et C_1' en un calcul C dans lequel les processeurs de G_0 se comportent comme dans C_0', et ceux de G_1 comme dans C_1'. Dans C, les processeurs de G_0 terminent avec $data = d_0$, et ceux de G_1 terminent avec $data = d_1$: contradiction. ∎

Nous considérons maintenant le cas des réseaux faiblement connexes. Contrairement à ce qu'une étude trop rapide pourrait laisser croire, il est possible de diffuser une donnée à partir de P_0 même si ce processeur est seul dans sa composante *fortement* connexe, du moins si les gardes d'émission sont disponibles ($CSP_{i/o}$). Considérons l'exemple ci-dessous, où l'on cherche à diffuser une donnée binaire dans le réseau G suivant

$$P_0 \longleftarrow P_1$$

```
P0 :: [ data = 0 -> P1?data0()
      ▮ data = 1 -> P1?data1() ]
P1 :: [ P0!data0() -> data := 0
      ▮ P0!data1() -> data :=1 ]
```

On vérifie aisément que $P = [\ P_0 \parallel P_1\ ]$ est un algorithme de diffusion (symétrique!) pour G. Cette utilisation non-standard des possibilités sémantiques de CSP est tirée d'une idée de Daniel Lehmann. En fait, cette utilisation semble avoir été pressentie depuis longtemps. En effet, dans [DS 80], Dijkstra et Scholten décrivent un modèle de réseau où les canaux sont unidirectionnels, mais peuvent cependant être utilisés en sens inverse pour transmettre des signaux exclusivement! Nous étudierons une généralisation (non modulaire) de cet exemple en section 6.2.3.4. Un tel comportement n'est pas possible en l'absence de gardes d'émission. Dans ce cas, en effet, P_0 est l'esclave de P_1, qui impose ses choix.

Théorème 6.1.2. *Le réseau G ci-dessus n'admet pas d'algorithme de diffusion en CSP_{in}.*

Preuve Soit P un tel algorithme, et soit C un calcul maximal où P_0 diffuse la donnée d_0. Soit p le nombre de communications (en fait émission de P_1 vers P_0) de C. Soit C_i^- le sous-calcul de C qui s'arrête juste avant l'émission i de P_1 vers P_0, et C_i^+ celui qui s'arrête juste après. Par convention C_0^+ est le calcul vide, et C_{p+1}^- est le calcul C entier.

Considérons maintenant la diffusion d'une autre donnée d_1 par P_0. Nous construirons ci-dessous un calcul D contenant p émissions de P_1 vers P_0, où, avec les mêmes conventions que ci-dessus, D_i^- et D_i^+ ont mêmes projections sur P_1 que C_i^- et C_i^+ respectivement pour tout i. Il est donc possible de construire un calcul où P_0 se comporte comme dans C alors que P_1 se comporte comme dans D. En particulier, à la terminaison, $data_0 = d_0$ alors que $data_1 = d_1$. Contradiction.

Cette construction se fait par récurrence sur i. Pour $i = 0$, nous prenons pour D_0^+ le calcul vide. Supposons D_i^+ construit. Remarquons que, à équivalence causale près, C_{i+1}^+ s'écrit $C_i^+; A_1; A_0; E$, où seul

P_0 est actif dans A_0 et P_1 dans A_1, et où E est l'émission $i+1$ de P_1 vers P_0 dans C. A la fin de A_1 (si $i < p+1$), P_1 est devant une commande d'émission vers P_0.

Nous pouvons donc construire $D_i^+; A_1$, puisque C_i^+ et D_i^+ ont même projection sur P_1. A la fin de ce calcul, P_1 est devant une commande d'émission vers P_0, qui ne peut être dans une garde. P_1 ne peut donc terminer sans émettre encore une fois au moins vers P_0. Ce calcul se prolonge en un calcul D terminé. D admet nécessairement un sous-calcul de la forme $D_i^+; A_1; B_0; F$, où seul P_0 est actif dans dans B_0, et où F est une émission de P_1 vers P_0. Le calcul D_{i+1}^- cherché est donc $D_i^+; A_0; B_0$. Il a par construction la même projection sur P_1 que C_{i+1}^-. A la fin de D_{i+1}^-, P_0 et P_1 sont prêts à établir leur communication $i+1$. Puisque la commande d'émission de P_1 ne peut être dans une garde, cette émission est nécessairement la même que l'émission $i+1$ dans C. Donc $E = F$. On prend alors comme D_{i+1}^+ le calcul $D_i^+; A_1; B_0; E$. ∎

6.2. Diffusion en $CSP_{i/o}$

Nous étudions dans cette section le problème de la diffusion modulaire en $CSP_{i/o}$. Nous montrons qu'il existe un algorithme de diffusion modulaire et symétrique pour les réseaux fortement connexes. On peut de plus exiger que le processeur initiateur détecte la terminaison de la diffusion.

6.2.1. Diffusion par front d'onde

L'idée maîtresse de notre solution est de considérer une stratégie de diffusion de l'information par "front d'onde". Cette stratégie est l'un des paradigmes classiques du domaine. Elle se retrouve par exemple dans [DS 80] et [WS 83]. Elle doit son nom à l'analogie avec le mécanisme de propagation des ondes au sein d'un milieu donné. Chaque point du milieu, une fois atteint par le front d'onde, se comporte comme un émetteur par rapport à elle. La superposition des émission de ces sources élémentaires forme le front d'onde, et provoque le déplacement observable macroscopiquement. Ici, l'onde correspond à l'apprentissage de la donnée diffusée par les processeurs. Les règles de propagation sont les suivantes.

Règles

1) Initialement, P_0 connaît la donnée d;
2) Chaque processeur est prêt à recevoir des données de chacun de ses voisins entrants;
3) Chaque processeur émet toutes ses données à tous ses voisins sortants.

Si le réseau est fortement connexe, il devrait être clair que tous les processeurs auront appris la donnée initialement possédée par P_0 au bout d'un temps fini. Remarquons qu'il suffit en fait que le processeur P_0 soit une racine du réseau.

Une implantation directe de ces règles est décrite en figure 6.2.1 à l'aide d'une maquette. Initialement, la variable $data_u$ est indéfinie, sauf pour P_0, où elle est initialisée à d, la donnée à diffuser. Le contenu de cette variable est mis à jour à chaque réception de message $new(x)$, et systématiquement envoyé à tous les voisins sortants par un message $new(data)$. L'émission est contrôlée par deux variables. ok_u est vraie si le processeur a été atteint par la vague (l'onde) de diffusion, ce qui garantit que $data_u$ est bien définie. $sent_u[v]$ est mise à vrai si P_u a envoyé un message $new(data)$ à P_v; ceci garantit que P_v ne sera pas submergé par les messages de P_u. Exactement un message est donc envoyé sur chaque canal. Chaque processeur sait donc qu'il ne doit attendre qu'un message sur chaque canal entrant. C'est le rôle de la variable $received_u[w]$ qui est mise à vrai lors de la réception par P_u d'un message $new(x)$ de P_w. Un processeur qui a échangé exactement un message sur chaque canal termine.

Théorème 6.2.1.1. *Il existe un algorithme de diffusion modulaire et symétrique en $CSP_{i/o}$ pour la famille des réseaux dont P_0 est racine.*

Preuve La modularité et la Σ_0-symétrie de l'algorithme presenté en figure 6.2.1 découlent des conditions suffisantes présentées aux chapitres 3 et 4. Nous fixons un réseau G dont P_0 est racine, et nous étudions l'algorithme P associé à ce réseau. Pour la correction de cet algorithme, on remarque que les prédicats suivants sont **invariants**, en ce sens qu'ils sont satisfaits dans tous les états où le contrôle dans chaque processeur est devant sa boucle externe, ou après dans le cas où le processus est terminé*.

* C'est le sens que nous donnerons à cette expression tout au long de ce chapitre sauf mention explicite du contraire.

```
P_u ::  [ RESET_u;
          *[ ▌ in ¬received[in]; P_in?new(x)
             -> data:= x; received[in]:= true; ok:= true
             ▌ out ok; ¬sent[out]; P_out!new(data)
             -> sent[out]:= true
          ]]
```

avec $RESET_u$ qui vaut si $u \neq 0$

```
received:= false; sent:= false; ok:= false;
```

et $RESET_0$ qui vaut

```
data:= d; received:= false; sent:= false; ok:= true;
```

NB: On rappelle que dans tous les programmes présentés dans cette thèse, l'affectation d'une constante v à toutes les cellules d'in tableau $tab[range]$ est notée $tab := v$ lorsqu'aucune confusion n'est à craindre.

Figure 6.2.1.

1) $sent_u[v] = received_v[u]$ (communication synchrone!);
2) si $u \neq 0$, ok_u si et seulement si $received_u[v]$ pour au moins un voisin entrant P_v; ok_0 est toujours vrai;
3) si $data_u$ est définie alors $data_u = d$;
4) P_u est terminé si et seulement si ok_u est vrai et $received_u[v]$ et $sent_u[w]$ sont vrais pour tous les voisins entrants P_v et sortants P_w.

Nous montrons que tout calcul C de P est fini. En effet, au plus un message est échangé sur chaque canal. Chaque processeur ne peut donc effectuer qu'un nombre fini d'itérations de sa boucle, et donc un nombre fini de transitions. Les projections de C sur tous les processeurs sont donc finies, et C est donc fini.

Considérons un calcul C maximal. C étant fini, considérons l'état final de C. Par maximalité chaque processus est devant sa boucle externe ou après elle. Les invariants ci-dessus s'appliquent donc. Montrons que, dans cet état, ok_u est vrai pour tout processeur P_u. On a trivialement ok_0 qui est vrai. Supposons que ok_u soit vrai pour un processeur P_u, et considérons un voisin sortant P_v de P_u. Supposons que ok_v soit faux. Alors nécessairement $received_v[u]$ est faux. Par l'invariant 1, $sent_u[v]$ est faux. Par l'invariant 4, P_u ne peut être terminé. Donc le contrôle dans P_u est nécessairement au sommet de sa boucle principale. De même, puisque $received_v[u]$ est faux, P_v ne peut pas être terminé, et le contrôle se trouve donc au sommet de sa boucle principale. Mais la garde de communication entre P_u et $P_v$est donc ouverte et C ne serait pas maximal. Donc, ok_v est vrai. L'ensemble des processeurs P_u tel que ok_u est vrai contient donc le cône de racine P_0 dans G. Comme P_0 est racine de G, il contient tous les processeurs de G, ce que nous voulions démontrer.

Supposons maintenant que P_u ne soit pas terminé dans l'état final de C. Le contrôle dans P_u est au sommet de sa boucle. Supposons que $sent_u[v]$ soit faux. Alors $received_v[u]$ est faux, et l'on conclut comme ci-dessus par maximalité de C. Donc, $sent_u[v]$ est vrai pour tout voisin sortant P_v. Nous avons montré plus haut que ok_u est vrai. Donc $received_u[v]$ est faux pour un certain voisin entrant P_v. Mais dans P_v, $sent_v[u]$ est faux et nous avons montré que ok_v est vrai. Donc P_v ne peut pas être terminé et l'on conclut encore par maximalité. ∎

L'algorithme précédent provoque l'échange d'exactement un message sur chaque canal. Sa complexité est donc en $O(E)$, où E est le nombre de canaux du réseau.

6.2.2. Détection de la terminaison

Un inconvénient de l'algorithme précédent est que le processeur initiateur, P_0, n'a aucun moyen de détecter la terminaison globale de la diffusion. Considérons par exemple le réseau G suivant

$$P_0 \longleftrightarrow P_1 \longleftrightarrow P_2 \longleftrightarrow P_3$$

Le scénario suivant peut se produire: P_0 envoie la donnée d à P_1, qui la renvoie à P_0 puis à P_2 *etc.* P_0 termine donc avant même que P_3 ait été atteint par la vague de diffusion, bien qu'il soit assuré qu'il le sera au bout d'un temps fini. Ce type d'exigence est crucial si la diffusion est en fait utilisée par P_0 pour collecter une information globale sur le réseau, comme c'est le cas dans [TL 86], ou bien dans les méthodes de détection de la terminaison répartie par "freezing" ([Francez 80]). L'algorithme que nous avons présenté peut effectivement être adapté en fonction de cette nouvelle spécification.

L'idée de base est de reprendre l'analogie avec la diffusion centrifuge d'une onde dans un milieu donné. Supposons maintenant que ce milieu soit borné. Lorsque le front d'onde atteint la frontière, une nouvelle onde, centripète cette fois, est engendrée, progressant en sens contraire de l'onde initiale, pour être finalement absorbée par la source. Cette stratégie est connue sous le nom de "écho", et a été utilisée par Dijkstra, Scholten et van Gasteren ([DFG 83]). En plus des messages de diffusion $new(x)$, les processeurs s'échangent des signaux $done()$ de terminaison, exprimant qu'ils ont terminé leur rôle dans la diffusion proprement dite. Chaque processseur reconnaît comme père le voisin entrant dont il a reçu son premier message $new(x)$. La règle est que le père est le dernier voisin entrant à recevoir le signal de terminaison. La relation de paternité détermine en fait un arbre recouvrant du réseau G, de racine P_0. Il devrait être donc clair que P_0 recevra le dernier message de terminaison, et donc le dernier message tout court, échangé dans le réseau.

Comme en section 6.1, les signaux de terminaison $done()$ utilisent les canaux en sens inverse. Ils sont gérés par deux variables $received1_u[v]$ et $sent1_u[v]$. La variable $father$ est utilisée pour conserver le nom du père. L'algorithme est présenté en figure 6.2.2. On remarquera l'abus de langage consistant à écrire

$$\mathop{\mathbf{I}}_{in} \ldots ;\ P_{father}!done() \text{ -> } \ldots$$

au lieu de

$$\mathop{\mathbf{I}}_{in} \ldots ;\ in = father\ ;\ P_{in}!done() \text{ -> } \ldots$$

On pourra vérifier dans la preuve que cet abus est bien justifié. Cet algorithme se présente en fait comme la superposition de deux diffusions, l'une centrifuge avec les messages $new(x)$, et l'autre centripète avec les signaux $done()$, cette dernière étant déclenchée par la terminaison (locale) de la première. Cette méthode de construction d'algorithmes par "superposition" est en fait un paradigme très général dans le domaine de l'algorithmique répartie (*cf.* par exemple [BT 83] ou [TL 86]). Elle ne semble pas avoir été étudiée d'un point de vue théorique jusqu'à présent. Nous ne savons pas, en particulier, comment déduire les propriétés de l'algorithme résultant à partir de celles des algorithmes composants. Cette méthode reste un sujet de recherche ouvert, et, en attendant, nous pouvons l'utiliser comme guide pour une preuve de correction "ad-hoc" (on pourra trouver un essai de synthèse dans [Gafni 86]).

Théorème 6.2.2.1. *Il existe un algorithme de diffusion modulaire et symétrique en $CSP_{i/o}$ pour les réseaux de racine P_0 tel que, dans tout calcul, P_0 échange le dernier message.*

Preuve Nous fixons un graphe G de racine P_0 et nous considérons le système P dérivé de ce graphe à partir de la maquette de la figure 6.2.2. Les invariants décrits en section 6.2.1 restent valables. Les invariants suivants sont aussi satisfaits.

5) $sent1_u[v] = received1_v[u]$;
6) ok_u est vrai si et seulement si $father_u$ est défini ou $u = 0$;
7) $received1_u[father]$ est vrai seulement si ok_u, $received_u$, $sent_u$, $received1_u$, $sent1_u$ sont vrais; P_u ne peut donc plus communiquer;
8) P_u est terminé si et seulement si ok_u, $received_u$, $sent_u$, $received1_u$, $sent1_u$ sont vrais.

```
P_u :: [ RESET_u
        *[ ▮ in ¬received[in]; P_in?new(x)
           -> if ¬ok then father:= in; ok:= true end;
              data:= x; received[in]:= true
           ▮ out ok; ¬sent[out]; P_out!new(data)
           -> sent[out]:= true
           ▮ out ¬sent1[out]; P_out!done()
           -> sent1[out]:= true
           ▮ in ok; ¬received1[in]; in ≠ father; P_in?done()
           -> received1[in]:= true
           ▮ u ≠ 0; ok; ¬received1[father];
              ⋀ in received[in]; ⋀ out sent[out];
              ⋀ in (in ≠ father ⟹ received1[in]); ⋀ out sent1[out];
              P_father?done()
           -> received1[father]:= true
        ]]
```

avec $RESET_u$ comme en figure 6.2.1, avec en plus

```
sent1:= false; received1:= false
```

Figure 6.2.2.

L'invariant 6 est crucial, puisqu'il garantit que les expressions booléennes contenant *father* sont bien définies. On montre comme en 6.2.1 que tous les calculs sont finis. Soit C un calcul maximal. On montre que C ne peut être en échec, et que dans son état final, ok_u est vrai pour tous les processus P_u grâce au fait que P_0 est racine de G. Donc, $father_u$ est défini pour tout $u \neq 0$. Définissons le graphe $FATHER_C$ de la manière suivante. Ses sommets sont ceux de G. Il existe un côté de P_u vers P_v si $father_v = u$ dans l'état final de C. Nous utilisons ci-dessous plusieurs lemmes qui seront démontrés séparément.

Lemme 6.2.2.1. *$FATHER_C$ est un arbre recouvrant de G, de racine P_0.*

Supposons que P_u ne soit pas terminé dans l'état final de C. Il est donc bloqué, ses variables ok_u, $received_u$ et $sent_u$ étant vraies. Le contrôle dans P_u est nécessairement au sommet de sa boucle externe. Supposons que $received1_u[v]$ soit faux, pour un certain voisin entrant P_v de P_u. Donc, par l'invariant 6, $sent1_v[u]$ est faux. Par l'invariant 8, P_v ne peut avoir terminé. Le contrôle dans P_v est donc aussi nécessairement au sommet de sa boucle principale. Si v n'est pas $father_u$, alors la communication est possible dans l'état final de C, et C n'est pas maximal. Donc $father_u = v$, et $u \neq 0$. De nouveau par maximalité de C, il existe un voisin sortant de P_u, disons P_w, tel que $sent1_u[w]$ soit faux.

En conclusion, dans l'état final de C, tout processeur P_u de P est soit terminé, soit bloqué en attente d'émission du signal *done*() vers un de ses voisins sortant dont il est le père. Considérons alors le graphe WFG_C ("Wait-For-Graph", *cf.* [BT 83]) construit de la maière suivante. Ses sommets sont ceux de G. Il existe un côté de P_u vers P_v si, dans l'état final de C, $sent1_u[v]$ est faux.

Lemme 6.2.2.2. *WFG_C est un sous-graphe de $FATHER_C$*

Lemme 6.2.2.3. *Si WFG_C a un côté, alors il a un cycle.*

Puisque $FATHER_C$ est un arbre par le lemme 6.2.2.1, on en déduit que WFG_C n'a aucun côté, et donc que tous les processus sont terminés dans l'état final de C. ∎

Preuve du lemme 6.2.2.1

Puisque ok_u est vrai pour tout processeur P_u différent de P_0, $father_u$ est défini par l'invariant 6 pour tous ces processeurs. D'autre part, si P_v est le père de P_u, alors P_u a reçu un message $new(x)$ de P_v, et donc il existe un côté de P_v vers P_u dans G. Donc $FATHER_C$ recouvre G, sauf peut-être P_0. Mais le premier message échangé est nécessairement émis par P_0, et donc P_0 est le père d'au moins un processeur. Dans ce graphe, tous les processeurs ont exactement un côté entrant, sauf P_0 qui n'en a aucun.

Supposons qu'il existe un cycle

$$P_{u_1} \longrightarrow \cdots \longrightarrow P_{u_p} \longrightarrow P_{u_1}$$

avec $p \geq 2$. Soit λ_i la transition où $P_{u_{i+1}}$ reçoit son premier message $new(x)$ de P_{u_i} (les indices sont pris modulo p). λ_i précède strictement λ_{i+1} dans le calcul C (c'est en fait une propriété satisfaite dans tous les calculs causalement équivalents à C). Donc, par transitivité, λ_1 précède strictement λ_1. Contradiction.

$FATHER_C$ n'a donc pas de cycle orienté. C'est donc un arbre orienté. Chaque sommet sauf P_0 a exactement un côté entrant. Donc seul P_0 peut en être racine. ∎

Preuve du lemme 6.2.2.2

Supposons que WFG_C ait un côté de P_u vers P_v. Alors dans l'état final de C, $sent1_u[v]$ est faux. D'après ce que nous avons vu, ceci implique que P_v a pour père P_u, et $u \neq 0$. Donc $FATHER_C$ a aussi un côté de P_u vers P_v. ∎

Preuve du lemme 6.2.2.3

Supposons que WFG_C ait un côté de P_u vers P_v. Alors $sent1_u[v]$ est faux. On montre comme dans le début de la preuve du théorème 6.2.2.1 qu' il existe un voisin sortant P_w de P_v tel que $sent1_v[w]$ soit faux. Donc WFG_C a aussi un côté de P_v vers P_w. WFG_C étant fini, il a donc un cycle au moins. ∎

6.2.3. Remarques

6.2.3.1. Efficacité

Dans l'algorithme de diffusion présenté en section 6.2.1 chaque processeur reçoit la valeur à diffuser de chacun de ses voisins entrants. A strictement parler, seule la première réception est informative. Pour les autres, seule la synchronisation associée à la réception est informative. On peut donc se contenter de recevoir une donnée vide dans ce cas, en modifiant la commande gardée par la réception en

```
if ¬ok then data := x; ok := true end;
received[in] := true
```

La donnée reçue est alors négligée, sauf lors de la première réception. Le problème est que le processeur émetteur ne peut, en général, déterminer si le processeur destinataire a déjà reçu la donnée, auquel cas il peut simplement ne pas tenter d'émettre la donnée. Un cas où il peut conclure est celui où il a déjà lui-même reçu la donnée du processeur destinataire. En ce cas, celui-ci sait aussi qu'il a déjà envoyé la donnée et ne doit donc pas en attendre d'écho. On remplace donc la garde d'émission par

```
▮ ok; (out ∈ In ⟹ ¬received[out]); ¬sent[out]; P_out ! new(data)
out
-> sent[out] := true
```

et celle de réception par

```
P_u :: [ RESET_u
         *[ ▮ in ¬received[in]; P_in?new(x)
            -> if ¬ok then father:= in; ok:= true end;
               data:= x; received[in]:= true
            ▮ out ok; out ≠ father; ¬sent[out]; P_out!new(data)
            -> sent[out]:= true
            ▮ u ≠ 0; ok; ¬sent[father]; DONE(father); P_father!new(data)
            -> sent[father]:= true
         ]]
```

avec $DONE(father)$ qui vaut

$\bigwedge_{in} received[in]$; $\bigwedge_{out} (out \neq father \Longrightarrow sent[out])$

$RESET_u$ qui vaut pour $u \neq 0$

received:= **false**; *sent*:= **false**; *ok*:= **false**;

et $RESET_0$ qui vaut

data:= *d*; *received*:= **false**; *sent*:= **false**; *ok*:= **true**;

Figure 6.2.3.1.

```
▮ in (in ∈ Out ⟹ ¬sent[in]); ¬received[in]; P_in?new(x)
-> data := x ; received[in] := true ; ok := true
```

Cette optimisation revient en fait à considérer qu'un "non-messagé' est échangé entre deux processus qui savent qu'ils doivent échanger un message et que ce message est en fait inutile. Noter que si un seul des processus est conscient de ce fait, le refus de communiquer conduirait à un blocage plutôt qu'à un gain de temps ! Cette optimisation sera particulièrement sensible si la donnée à diffuser a une taille importante, et si le réseau a de nombreux canaux bidirectionnels. Remarquons qu'elle ne peut empêcher un certain degré de redondance dans les transferts d'information. Il semble que cette remarque soit générale. La modularité et la symétrie imposées à l'algorithme font que celui-ci ne peut utiliser le réseau de manière optimale. Il doit mettre en oeuvre des procédures "tolérantes", nécessairement plus coûteuses.

6.2.3.2. Réseaux bidirectionnels

Dans le cas d'un réseau bidirectionnel, les deux algorithmes étudiés peuvent se confondre en un seul, dans lequel les les messages *new*(*data*) jouent le rôle de signaux. L'algorithme ainsi modifié est présenté en figure 6.2.3.1 (remarquer que $In = Out$).

Aucun message n'est échangé dans le réseau après la terminaison de P_0. Remarquons que l'optimisation présentée en 6.2.3.1 s'applique aussi dans ce cas.

6.2.3.3. Diffusion d'une suite de données

Considérons maintenant le problème de la diffusion par P_0 d'une suite de données produites dynamiquement. Il suffit pour cela d'utiliser une itération répartie de l'algorithme de diffusion d'une donnée présenté en figure 6.2.1. Cette itération est bien synchronisée au sens du chapitre 5, puisque exactement un message est échangé sur chaque canal au cours de toute exécution. Ceci résoud donc le problème dans le cas d'une suite infinie de données. Tous les processeurs recevront les même valeurs, dans le même ordre. Remarquons que la condition de synchronisation ci-dessus montre que l'avance des processeurs les uns par rapport aux autres est bornée. Soit δ le diamètre du réseau. Si le processeur P_u a reçu p données alors tous les processeurs ont au moins reçu les $(p - \delta)$ premières données. Cette estimation est grossière, comme le montre le cas d'un anneau: si un processus a reçu p données, alors tous les processus ont reçu les $(p - 1)$ premières données, et ce quelle que soit la taille de l'anneau.

Théorème 6.2.3.1. *Il existe un algorithme modulaire et symétrique en $CSP_{i/o}$ pour la diffusion d'une suite de données dans les réseaux fortement connexes.*

Supposons maintenant qu'au lieu de diffuser un flot continu de données, P_0 ne veuille diffuser qu'une suite finie de données et ensuite forcer la terminaison. Il lui suffit pour cela de diffuser une donnée supplémentaire, un marqueur de fin de diffusion en quelque sorte. Avant d'initialiser une nouvelle phase, chaque processeur teste si la dernière donnée diffusée est égale au marqueur et si c'est le cas, il termine. Chaque processus (sauf le processus initiateur) est donc de la forme suivante :

```
Q_u ::  [ data := null
          *[ data ≠ ''end_of_data'' -> P_u ]
        ]
```

D'après la discussion ci-dessus, la condition d'arrêt est satisfaite pour un processeur au début d'une phase si et seulement si elle est satisfaite pour tous les processeurs au début de cette même phase. Elle est donc synchronisée, et d'après les théorèmes généraux vus au chapitre 5 ceci garantit la terminaison propre de l'algorithme.

6.2.3.4. Réseaux faiblement connexes

L'idée est de généraliser la solution présentée en section 6.1 dans le cas de deux sommets. La difficulté est le traitement de la variable *ok*, qui exprime que le processeur connaît la donnée diffusée. Dans le cas des réseaux fortement connexes, on peut considérer que *ok* devient vrai à la première réception qui doit précéder toute émission. Dans le cas des réseaux faiblement connexes, on ne peut privilégier les réceptions par rapport aux émissions, puisque l'une et l'autre peuvent correspondre à une réception de connaissance où à une émission. Un autre critère est donc nécessaire. Nous adoptons ici un critère non modulaire (mais symétrique) qui est la distance $dist(u, 0)$ d'un processeur P_u à la racine dans le réseau non orienté sous-jacent. L'idée est que la diffusion se fait de manière centrifuge à partir de P_0. La variable *ok* prend la valeur true lors de la première communication avec un processeur "plus proche" de P_0, qui doit précéder tout autre communication. L'algorithme est présenté en figure 6.2.3.2.

Un algorithme de diffusion modulaire mais non symétrique pour la famille des réseaux faiblement connexes centrifuges peut être déduit de l'algorithme de diffusion en CSP_{no} présenté en section 6.4.2. Nous ne savons pas s'il existe une solution en $CSP_{i/o}$ à la fois modulaire et symétrique à ce problème.

6.3. Diffusion en CSP_{in}

Nous avons étudié en section 6.2 le problème de la diffusion en $CSP_{i/o}$. Nous avons montré l'existence d'un algorithme modulaire et symétrique pour les réseaux de racine P_0. Grâce à l'utilisation des gardes d'émission, cet algorithme est simple et élégant. De fait, le cas de CSP_{in} est beaucoup plus complexe. La solution ci-dessus peut s'étendre au cas de CSP_{in} au prix d'une localisation de la vague de diffusion. L'algorithme ainsi obtenu est pratiquement séquentiel, en ce sens qu'au plus un processeur n'est pas en attente à chaque instant. De plus, la terminaison ne peut être assurée que dans le cas des réseaux bidirectionnels. Par contre, si la symétrie ou la modularité est abandonnée, on peut obtenir des algorithmes de diffusion pour les réseaux de racine P_0 unidirectionnels.

```
P_u ::  [ RESET_u
          *[ ▮_in (CLOSER(in,u) ∨ ok); ¬received[in]; P_in?new_0()
             -> data:= 0; received[in]:= true; ok:= true
             ▮_in (CLOSER(in,u) ∨ ok); ¬received[in]; P_in?new_1()
             -> data:= 1; received[in]:= true; ok:= true
             ▮_out (CLOSER(out,u) ∨ ok); ¬sent[out]; P_out?new_0()
             -> data:= 0; sent[out]:= true; ok:= true
             ▮_out (CLOSER(out,u) ∨ ok); ¬sent[out]; P_out?new_1()
             -> data:= 1; sent[out]:= true; ok:= true
          ]]
```

avec $CLOSER(v,u)$ qui vaut $(dist(v,0) < dist(u,0))$

Figure 6.2.3.2.

6.3.1. Pseudo-diffusion modulaire et symétrique

La stratégie du font d'onde appliquée en section 6.2.1 ne peut être directement transposée au cas de CSP_{in}. En effet, cette stratégie conduit à une multitude de micro-diffusions concurrentes autour de chaque processeurs du réseau. Ces micro-diffusions entrent inévitablement en conflit les unes avec les autres. Du fait de la symétrie, ces conflits doivent nécessairement être résolus dynamiquement. La combinaison de gardes d'émission et de gardes de réception permet précisément un tel traitement comme le montre l'exemple suivant.

$P = [P_a \parallel P_b]$

```
P_a ::  [ sent_a:= false; received_a:= false;
          *[ ¬sent_a; P_b!pebble() -> sent_a:= true
          ▮ ¬received_a; P_b?pebble() -> received_a:= true
        ]]
P_b ::  [ sent_b:= false; received_b:= false;
          *[ ¬sent_b; P_a!pebble() -> sent_b:= true
          ▮ ¬received_b; P_a?pebble() -> received_b:= true
        ]]
```

L'utilisation des gardes de réception seulement ne permet pas ce type de négociation dynamique entre processeurs. Les processeurs doivent décider, sur la base de leur connaissance locale uniquement, de l'opportunité d'émettre un message; une fois la décision prise, elle est irrévocable!

La stratégie du front d'onde doit être adaptée pour éviter, de manière structurelle, de tels conflits. Le moyen le plus simple est de localiser la progression du front d'onde. A chaque instant, seul un processeur est "actif", en ce sens qu'il procède à une micro-diffusion. Tous les autres processeurs sont alors passifs, dans divers états. Contrairement à l'algorithme de la figure 6.2.1, le contrôle est maintenant *centralisé* à chaque instant en un processeur précis. Ce processeur varie au cours du calcul, le passge du contrôle d'un processeur à l'autre se faisant au moyen de messages explicites.

Ces considérations nous conduisent en fait à des stratégies bien connues, celles de recherche dans les graphes en profondeur d'abord ("depth-first") ou en largeur d'abord ("breadth-first"). Les deux peuvent ici être utilisées; il est cependant classique que la première est la plus simple à implanter dans le cas de réseaux finis. C'est donc elle que nous choisissons ici. La diffusion à laquelle elle conduit est parfaitement séquentielle et centralisée. Par rapport à la solution entièrement répartie et parallèle, on constate une perte

Pour $u \neq 0$

```
P_u ::  [ received:= true; sent:= false; ok:= false;
(1)       *[ ▮ in ¬received[in]; P_in?new(x, w)
             -> received[in]:= true;
                if ¬ok
                then ok:= true; data:= x; wait:= w ∪ {u};
                     *[ ▮ out ¬sent[out]; out ∉ wait
(2)                     -> P_out!new(data, wait);
                           sent[out]:= true;
(3)                        P_out!done() ]
                end
(4)             P_in?done()
          ]]

P_0 ::  [ data:= d; sent:= false; wait:= {0};
          *[ ▮ out ¬sent[out]
(2)          -> P_out!new(data, wait);
                sent[out]:= true;
(3)             P_out!done()
          ]]
```

Figure 6.3.1.

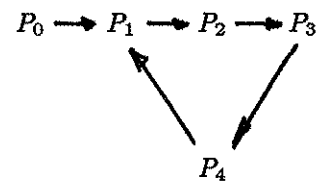
en temps, puisque chaque proceseur est actif à tour de rôle. Ces caractéristiques apparaissent comme le prix incompressible de l'absence de gardes d'émission.

La stratégie de recherche en profondeur d'abord doit être adaptée pour tenir compte de la présence éventuelle de cycles dans le réseau. Contrairement à ce qui se passe en programmation centralisée classique, il n'est pas possible ici de "marquer" les processeurs afin de garder traces des visites déjà effectuées. Il n'est pas possible pour un processeur de "demander" à un autre si celui-ci a déjà été visité sans lui envoyer un message, et donc risquer un blocage. Cette information doit donc être diffusée explicitement en même temps que la donnée. La perte en temps se double donc d'une perte en espace. La gestion des conflits, assurée auparavant par le médium de communication, doit maintenant être rendue explicite, ce qui nécessite donc l'acheminement d'informations supplémentaires.

L'algorithme résultant de ces considérations est décrit en figure 6.3.1. Pour plus de clarté, on utilise ici une maquette d'un forme non conforme à celle décrite en section 3.3. Ceci permet de différencier plus aisément les communications placées dans les gardes et celles qui ne le sont pas. Il devrait cependant être clair que cet abus de notation, utilisé aussi dans le reste de ce chapitre, se réduit facilement à une véritable forme normale. Comme en section 6.2.2, les tableaux *recieved*$[In]$ et *sent*$[Out]$ gèrent respectivement les émissions et réceptions. Au plus un message est échangé sur chaque canal. Le booléen *ok* dénote que le processeur a été atteint par la vague de diffusion. Les messages $new(data, wait)$ constituent la vague de diffusion proprement dite, alors que les signaux $done()$ constituent les échos. Ces signaux garantissent la localisation du front d'onde en imposant la séquentialité du parcours du réseau. L'ensemble *wait* contient les noms des processus en attente.

Cet algorithme, comme celui de la section 6.2.2, réalise correctement la diffusion de la donnée d. Tous les calculs sont finis. Dans l'état final, chaque processus P_u possède dans sa variable $data_u$ la donnée d diffusée par P_0. P_0 échange le dernier message de tout calcul. Cependant, ce n'est pas un algorithme de

diffusion. En effet, la terminaison propre des calculs n'est pas garantie, comme le montre l'exemple suivant.



Il est facile de vérifier que l'algorithme se bloque, les processeurs P_0, P_1, P_3 et P_4 ayant atteint leur état final, mais P_2 étant en attente d'un message de P_4. Un algorithme P vérifiant les conditions suivantes sera appelé un algorithme de **pseudo-diffusion.**

1) Tous les calculs de P sont finis.
2) Si, dans l'état initial, $data_0 = d$, dans l'état final $data_u = d$ pour tout processus P_u.

Nous pouvons alors montrer le théorème suivant.

Théorème 6.3.1.1. *Il existe un algorithme modulaire et symétrique de pseudo-diffusion en CSP_{in} pour les réseaux de racine P_0, tel que P_0 termine proprement et échange le dernier message de tout calcul.*

Preuve Fixons un réseau G de racine P_0, et considérons l'algorithme P associé. Nous remarquons d'abord qu'au plus un message est échangé sur chaque canal. Les calculs sont donc finis.

Nous dirons qu'un processeur P_u est **passif** si le contrôle dans P_u est au sommet (1) de sa boucle, ou si P_u est terminé. P_u est **en attente** si le contrôle dans P_u est entre les émissions (2) et (3). Nous dirons sinon que P_u est **actif.** On vérifie que les propriétés suivantes sont invariantes.

1) Au plus un processeur est actif.
2) Quand P_u est actif, l'ensemble $wait_u$ reçu en (1) est exactement l'ensemble des processeurs en attente.
3) $received_u[v] = sent_v[u]$.

Soit WFG le sous-graphe de G qui a un arc de P_u vers P_v si P_u est en attente sur une émission (3) $P_v!done()$.

4) WFG n'a pas de cycle. Donc WFG est nécessairement une chaîne de racine P_0 et d'extrémité le seul proceseur actif P_u.

Soit C un calcul maximal de P. Considérons l'état final de C, et la graphe d'attente WFG_C associé. S'il a au moins un côté, soit P_u l'unique processeur actif. Par maximalité, P_u est nécessairement bloqué en (2) ou en (4). P_u ne peut être bloqué en (4), puisque le processeur précédent dans WFG est nécessairement bloqué en (3). Donc, P_u est bloqué en (2), en attente d'émission d'un message $new(x, w)$ vers un processeur P_v. Nécessairement, $sent_u[v]$ est faux, et $v \notin wait_u$. Donc, par l'invariant (3), P_v n'est pas en attente sur (2), et donc P_v est passif. Le contrôle dans P_v est donc en (1), et la communication peut donc avoir lieu, contrairement à la maximalité de C. Donc, dans l'état final de C, WFG_C est vide, et tous les processeurs sont passifs. En particulier, P_0 ne peut être que terminé.

Montrons que ok_u est alors vrai pour tout processeur P_u. ok_0 est vrai. Supposons que ok_u soit vrai. Alors, P_u a été actif. Nous avons vu que, dans l'état final, il est passif. Il ne peut redevenir passif avant d'avoir activé tous ses voisins sortants P_v, avec $v \notin wait_u$. D'autre part, un proceseur P_v tel que $v \in wait_u$ a nécessairement déjà été activé puisqu'il est en attente. Donc, puisque dans l'état final tous les processeurs sont passifs, tous les voisins sortants de P_u auront mis leur variable ok_v à vrai dans l'état final de C. Puisque P_0 est racine de G, ok_u sera donc vrai pour tous les processeurs P_u de G dans l'état final de C. ∎

Nous nous trouvons ici devant le phénomène dit de terminaison répartie, que nous étudierons en détail au chapitre 8. P_0 détecte effectivement la fin de la terminaison, et, en fait, détecte le blocage de l'algorithme. Cependant, vu les restrictions de symétrie et de modularité que nous considérons, P_0 n'a aucun moyen de forcer la terminaison de l'algorithme. En effet, le seul moyen éventuel serait de provoquer la diffusion d'une nouvelle vague comportant un marqueur de terminaison, mais cette nouvelle vague ne terminerait pas elle-même! Le problème vient ici de l'absence de gardes d'émission qui implique une gestion explicite des conflits, liée à un trop faible degré de connexité du réseau. Il n'est pas possible de diffuser de diffuser l'information de manière suffisante pour permettre aux divers processus de provoquer la terminaison synchronisée de l'algorithme.

Pour $u \neq 0$

```
P_u ::
(1)     [ ▮ P_in ! new(x, m)
          in
(2)       -> data:= x; marked:= m ∪ {u}
(3)          *[ ▮ out ∉ marked
               out
(4)             -> P_out ! new(data, marked)
(5)                P_out ? done(m)
(6)                marked:= m
             ]
(7)          P_in ! done(marked)
        ]

P_0 ::
(1,2)   [ data:= d; marked:= {0}
(3)       *[ ▮ out ∉ marked
             out
(4)          -> P_out ! new(data, marked);
(5)             P_out ? done(m);
(6)             marked:= m
(7)       ]]
```

Figure 6.3.2.

6.3.2. Diffusion dans les réseaux bidirectionnels

Il faut donc rajouter des conditions concernant le degré de connexité du réseau pour permettre une diffusion suffisante de l'information, et ainsi la synchronisation des terminaisons des divers processus. En l'absence de gardes d'émission, les chemins au long desquels circule l'imformation permettant la terminaison doivent pouvoir être déterminés localement. Une possiblité est d'utiliser précisément les chemins créés par la circulation de l'information liée à la diffusion proprement dite.

Le problème dans l'algorithme précédent est qu'un proceseur ne peut savoir si un processeur voisin a déjà été atteint par la diffusion lors de l'exploration d'une branche sœur dans le réseau. Grâce aux canaux bidirectionnels, nous pouvons mettre cette information à la disposition des processus actifs en l'accumulant dans une variable *marked* transmise au cours de l'exploration. De plus, grâce à cette information, le processeur actif pourra se contenter d'envoyer exclusivement des messages aux processeurs non encore explorés. Chaque processeur ne recevra donc qu'un seul message dans un état passif. L'algorithme résultant est présenté en figure 6.3.2 (remarquer que $In = Out$).

Théorème 6.3.2.1. *Il existe un algorithme de diffusion modulaire et symétrique en CSP_{in} pour les réseaux bidirectionnels de racine P_0.*

Preuve Soit G un tel réseau, et P l'algorithme associé. Il est facile de vérifier que tous les calculs de P sont finis. Un processus est **passif** s'il est devant (1) ou après (7), **en attente** s'il est entre (4) et (5), et **actif** sinon. Initialement, seul P_0 est actif. On montre qu'il y a au plus un processus actif dans chaque état du système.

On montre ensuite que si P_u est devant (3) ou devant (7) (et donc actif), alors $marked_u$ est exactement l'ensemble des processus qui ont été au moins une fois actifs.

Soit C un calcul maximal de P. On définit, comme en section 6.3.1, le graphe d'attente WFG d'un état. Il possède un arc de P_u vers P_v si P_u est en attente devant (5) dans ce état, devant une réception de P_v. WFG est un sous-graphe de G.

Montrons que WFG n'a pas de cycle. Supposons qu'il existe un état de C tel que le graphe d'attente associé ait un cycle.

$$P_{u_0} \longrightarrow P_{u_1} \longrightarrow \cdots \longrightarrow P_{u_p}$$

Soit λ_i la transition où P_{u_i} envoie le message $new(data, wait)$ à $P_{u_{i+1}}$. Nécessairement, λ_i précède strictement λ_{i+1} dans C. Donc λ_0 précède strictement λ_0. Contradiction.

Soit C un calcul maximal de P. Il est fini. Considérons son état final. Soit WFG_C le graphe d'attente de cet état. Soit P_u une feuille de WFG_C. P_u ne peut être passif ni en attente. Il est donc, par maximalité, bloqué devant (4) sur une émission vers P_v, avec $v \notin marked_u$. Donc, nécessairement, P_v est bloqué devant (1), ce qui contredit la maximalité. Donc WFG_C n'a pas de feuille. Il est donc vide, et tous les processus sont passifs dans l'état final de C. Si P_u a passé (1) et est redevenu passif, alors tous ses voisins sortants sont dans $marked_u$. Ils ont donc aussi passé (1). Puisque P_0 a passé (1) et qu'il est racine de G, tous les processus ont passé (1) dans l'état final de C. Ils sont donc après (7), et donc terminés.

On montre que si $data$ est défini, alors $data = d$. A la terminaison, $data$ est nécessairement défini. Donc $data_u = d$ pour tous les processus P_u. ∎

L'analyse présentée ici est en fait fondée sur la notion de *connaissance* . Le problème est de permettre aux processeurs de partager assez de connaissances pour pouvoir gérer leur terminaison de manière consistante. Cependant, le passage d'une formulation abstraite de l'algorithme en ces termes à la programmation concrète reste encore empirique et mal connu. On pourra trouver une approche de ce problème dans [HMR 85] et surtout [KT 86]. D'autre part, la complexité de cet algorithme est $O(N)$, où N est le nombre de processeurs dans le réseau, alors que celle de l'algorithme présenté en section 6.2.1 est $O(E)$, où E est le nombre de canaux dans le réseau. Il est donc plus efficace en terme de messages échangés. Par contre, il est implicitement séquentiel alors que celui de la section 6.2.1 était hautement parallèle. Tous deux sont symétriques et modulaires. Il est donc clair que nos outils ne sont pas assez précis pour mesurer le degré de parallélisme des algorithmes.

6.3.3. Diffusion symétrique

Nous avons vu plus haut que l'absence de gardes d'émission devait être compensée par des conditions plus restrictives quant au degré de connexité sur le réseau si l'on veut obtenir un algorithme modulaire et symétrique. Une autre possibilité est d'abandonner l'une de ces exigences. Nous étudions ici le cas d'un algorithme symétrique mais non modulaire. Le cas modulaire mais non symétrique sera étudié en section 6.4.2.

Théorème 6.3.3.1. *Il existe un algorithme de diffusion symétrique en CSP_{in} pour les réseaux de racine P_0.*

La stratégie est la suivante. Puisque la modularité n'est plus exigée, P_0 peut disposer d'une image globale du réseau pour guider la diffusion. Il choisit alors un arbre recouvrant T du réseau G, de racine P_0. Un tel arbre existe puisque P_0 est racine. Il envoie alors à tous ses fils dans T le message $new(d, T)$, et termine. Pour leur part, chaque processeur P_u, $u \neq 0$, est en attente d'un message $new(x, t)$ de l'un quelconque de ses voisins entrants. Sur réception d'un tel message, il le propage à chacun de ses fils dans t (s'il en a !), affecte à sa variable $data$ la donnée d et termine. L'algorithme résultant est décrit en figure 6.3.3.1.

Le prédicat $FATHER(u, v, t)$ est satisfait si le sommet v est un fils du sommet u dans l'arbre t. La fonction $FIND\_SPANNING\_TREE(g)$ appliquée à un graphe g produit, de manière non déterministe, un arbre recouvrant de g de racine P_0. Notons qu'il est crucial pour la symétrie du système que *tout* arbre recouvrant puisse être ainsi choisi. Remarquons que si σ est un automorphisme de G laissant stable P_0 et si T est un arbre recouvrant de G avec P_0 pour racine, alors $\sigma(T)$ l'est aussi. La symétrie de l'algorithme en découle.

Une autre solution, sans doute moins grossière, est de propager la vague de manière monotone, en n'autorisant que les émissions strictement centrifuges. Pour cela, on considère le prédicat $CLOSER(u, v)$ qui est satisfait si le sommet u est strictement plus proche de P_0 que le sommet v dans G. Un processeur P_u n'accepte initialement de message que des voisins entrants P_v tels que $CLOSER(v, u)$. Une fois l'un de ces messages reçu, il propage la valeur à ses voisins sortants P_w tels que $CLOSER(u, w)$ et accepte de la recevoir des voisins entrants P_v restants tels que $CLOSER(v, u)$. L'algorithme est présenté en figure 6.3.3.2.

Pour $u \neq 0$

```
P_u ::  [ sent:= false;
          [ ▌ P_in?new(x,t)
            in
            -> data:= x;
               *[ ▌ ¬sent[out]; FATHER(u,out,t)
                  out
                  -> P_out!new(data,t); sent[out]:= true
               ]
          ]]

P_0 ::  [ data:= d; sent:= false; T:= FIND_SPANNING_TREE(G);
          *[ ▌ ¬sent[out]; FATHER(0,out,T)
             out
             -> P_out!new(data,T); sent[out]:= true
          ]]
```

Figure 6.3.3.1.

```
P_u ::  [ RESET_u
          *[ ▌ ¬received[in]; CLOSER(in,u); P_in?new(x)
             in
             -> received[in]:= true; data:= x; ok:= true
             ▌ ok; ¬sent[out]; CLOSER(u,out)
             out
             -> P_out!new(data); sent[out]:= true
          ]]
```

avec $RESET_u$ qui vaut pour $u \neq 0$

received:= false; *sent*:= false; *ok*:= false

et $RESET_0$ qui vaut

data:= *d*; *received*:= false; *sent*:= false; *ok*:= true

Figure 6.3.3.2.

La preuve de correction de cet algorithme est moins triviale que celle de l'algorithme précédent. En fait l'absence de blocage est garantie par la propriété suivante du prédicat *CLOSER*. La fermeture transitive de ce prédicat est non réflexive. Le prédicat *CLOSER* joue donc exactement le rôle de l'ensemble *marked* de la section 6.3.2. Cependant, il est déterminé statiquement au lieu d'être construit dynamiquement. La symétrie de l'algorithme est garantie par la remarque suivante. Si σ est un automorphisme de G laissant P_0 invariant,

$$CLOSER(u,v) = CLOSER(\sigma(u),\sigma(v))$$

Remarquons que cet algorithme est plus économique que le précédent pour la longueur des messages échangés, quoique ceux-ci soient plus nombreux. Par ailleurs, l'information globale nécessaire est ici répartie entre tous

les processeurs, au lieu d'être centralisée en P_0. L'une ou l'autre de ces approches peuvent se justifier suivant les problèmes concrets considérés.

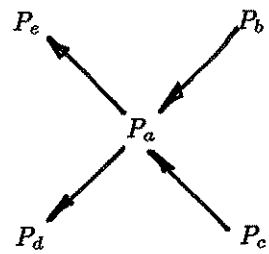
6.4. Diffusion en CSP_{no}

Comme on peut s'y attendre, le cas de CSP_{no} apparaˆcomme encore plus restrictif que celui de CSP_{in}. Dans ce dernier cas, l'existence d'algorithmes de diffusion modulaire et symétrique n'était garantie que dans le cas bidirectionnel. Nous allons ici montrer qu'il n'existe pas d'algorithme de diffusion modulaire et symétrique en CSP_{no} pour les graphes bidirectionnels de racine P_0. Par contre, si seule la modularité est exigée, il existe un algorithme s'appliquant aux réseaux de racine P_0.

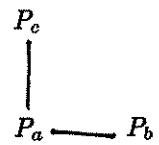
6.4.1. Non existence

Théorème 6.4.1.1. *Il n'existe pas d'algorithme de diffusion modulaire et symétrique en CSP_{no} pour les réseaux de racine P_0, même bidirectionnels.*

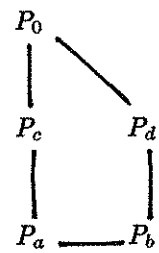
Preuve Supposons qu'un tel algorithme existe. Les processus sont déterminés par l'environnement local des processeurs associés. On peut donc identifier un processus par un nom de sommet, un ensemble de voisins entrants et un ensemble de voisins sortants. Par exemple, P_s avec $s = \langle a, \{b, c\}, \{d, e\}\rangle$ sera le processus associé au processeur P_a dans l'environnement local



Considérons le processus P_s avec $s = \langle a, \{b, c\}, \{b, c\}\rangle$

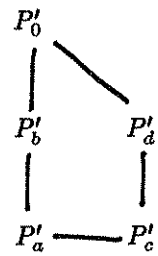


Considérons maintenant le réseau G ci-dessous



et l'algorithme P associé. Il est facile de vérifier que, dans tout calcul maximal C, P_a échange au moins un message (théorème 6.1.1).

Montrons que l'on peut supposer sans perte de généralité qu'il existe un calcul où P_a échange son premier message avec P_b. Deux cas se présentent. Soit il existe un calcul C où le premier message de P_a est échangé avec P_b et c'est le calcul cherché. Par le théorème 6.1.1, P_a échange au moins un message dans tout calcul terminé.Donc il existe un calcul où P_a échange son premier message avec P_c. Par commutation et restriction, il existe donc un calcul partiel C_0 tel que seul P_a soit actif et progresse jusqu'à devenir prêt à communiquer avec P_c. Puisque nous sommes en CSP_{no}, cette communication ne peut être dans une garde. Considérons alors le système P' construit sur le graphe G' suivant



Par modularité, $P'_a = P_a$. C_0 est aussi un calcul de P', puisque seul $P_a = P'_a$ y est actif. Dans l'état final de C_0, P'_a est devant une commande de communication avec P'_c qui ne peut être dans une garde. C_0 se prolonge en un calcul maximal C' et, par définition, P'_a termine dans C'. Donc dans C', P'_a échange son premier message avec P'_c. Sans perte de généralité, nous pouvons donc supposer que, dans un certain calcul C_a, P_a échange son premier message avec P_b. On remarquera que cette partie de la preuve ne dépend que de la modularité.

Supposons que c'est, par exemple, une réception. Par symétrie, il existe un calcul C_b où P_b échange son premier message, une réception, avec P_a. Par commutation et restriction, il existe un sous-calcul C'_a de C_a où seul P_a est actif et progresse jusqu'à sa première communication, une réception de P_b, qui ne peut être dans une garde. De même, il existe un sous-calcul C'_b de C_b où seul P_b est actif et progresse jusqu'à première communication exclusivement, une réception de P_a qui ne peut être dans une garde.

Soit alors C_1 la fusion de C'_a et C'_b. Dans l'état final de C_1, le contrôle dans P_a et P_b est devant une commande de réception, respectivement de P_b et P_a, qui ne peut être dans une garde. C_1 ne peut donc se prolonger en un calcul terminé. Contradiction. ∎

Ce théorème peut être justifié de la même façon que le théorème 6.3.1.1. L'absence des gardes de communication oblige à résoudre explicitement tous les conflits. La modularité entraine que l'acheminement des informations nécessaires doit se faire "à l'aveuglette" avant même d'avoir résolu les conflits, tandis que la symétrie provoque inévitablement l'apparition de ces mêmes conflits. Le problème est donc beaucoup plus profond qu'en CSP_{in} où l'obstacle est plutôt dû à un défaut de synchronisation dans la négociation des conflits qu'à l'impossibilité de leur résolution.

6.4.2. Diffusion modulaire en CSP_{no}

D'après le théorème précédent, la seule possibilité est d'oublier l'une des deux exigences de modularité ou de symétrie. Dans cette section, nous étudions l'existence d'une solution modulaire, nécessairement non symétrique en général.

Nous reprenons la stratégie fondamentale du front d'onde. En fait, le seul problème est de déterminer, par des considérations exclusivement locales, l'ordre dans lequel les différentes communications doivent être établies, de manière à éviter les conflits. En d'autres termes, il faut permettre à chaque processeur d'établir un ordre de priorité entre ses voisins. Le processeur communique d'abord avec ses voisins les plus prioritaires, que cette communication soit une réception ou un émission. Il est clair que les ordres de priorité des différents processeurs doivent être consistants. Ils doivent en fait être dérivés d'un ordre global sur les processeurs du réseau. Cette ordre ne peut être quelconque. En effet, la diffusion doit se faire de manière *centrifuge* à partir de la source P_0. Il est donc nécessaire que les processeurs "proches" de P_0 aient une priorité plus grande que ceux situés "loin" de la source. Considérons par exemple le réseau suivant :

$$P_0 \longleftrightarrow P_a \longleftrightarrow P_b \longleftrightarrow P_c$$

Supposons que c ait priorité sur a du point de vue de b. Alors le processeur P_b communiquera avec P_c avant de communiquer avec P_a et donc avant de recevoir la donnée à diffuser. P_c ne sera donc pas atteint par la vague de diffusion. Nous sommes ainsi conduit à la définition suivante. Soit $<$ un ordre total sur l'ensemble $Name$ des noms de processeurs.

Définition 6.4.2.1. *Un réseau G est dit centrifuge (pour l'orde $<$) si, pour tout processus P_u de G, il existe un chemin de P_0 vers P_u où les processeurs apparaissent en ordre décroissant.*

La stratégie de choix est donc la suivante. Soit $u \longrightarrow v$ un canal du réseau centrifuge G. Supposons d'abord qu'il n'y ait pas de canal $v \longrightarrow u$. Alors P_u propose une émission vers P_v si P_u a échangé un

```
P_u :: [ RESET_u
         *[ ▌ RTS(v)
            v
            -> P_v!new(data); exchanged[v]:= true
            ▌ DTR(v)
            v
            -> P_v?new(x)
               if x ≠ null then data:= x end
               exchanged[v]:= true
         ]]
```

avec $RESET_u$ qui vaut pour $u \neq 0$

```
data:= null; exchanged:= false;
```

et P_0 qui vaut

```
data:= d; exchanged:= false;
```

Figure 6.4.2.

message sur tous les canaux le liant à un voisin P_w, avec $w > v$. Le cas d'un canal entrant est similaire.Dans le cas d'un canal bidirectionnel entre P_u et P_v, le processeur de plus haute priorité émet et celui de plus basse priorité reçoit, l'autre canal étant alors inutilisé.

Nous définissons donc un tableau booléen *exchanged*. $exchanged_u[w]$ est vrai si P_u a échangé un message avec P_w. Ce tableau est initialisé à *false*. Nous pouvons définir le prédicat $RTS(v)$ ("Ready To Send") de la manière suivante.

$$\begin{aligned}&(\forall w \in (In \cup Out) \quad (w > v \Longrightarrow exchanged[w]))\\ &\wedge (v \in (In \cap Out) \Longrightarrow (u > v))\\ &\wedge (v \in Out)\\ &\wedge (\neg exchanged[v])\end{aligned}$$

De même, nous définissons $DTR(v)$ ("Data Terminal Ready") de la manière suivante

$$\begin{aligned}&(\forall w \in (In \cup Out)(w > v \Longrightarrow exchanged[w]))\\ &\wedge (v \in (In \cap Out) \Longrightarrow (u < v))\\ &\wedge (v \in In)\\ &\wedge (\neg exchanged[v])\end{aligned}$$

L'algorithme implantant la stratégie ci-dessus est présenté en figure 6.4.2.

On remarquera le léger abus d'écriture, puisque, par exemple, $P_v!new(d)$ n'a pas de sens si $v \notin Out$; mais $RTS(v)$ est toujours faux dans ce cas.

Théorème 6.4.2.1. *Il existe un algorithme de diffusion modulaire en CSP_{no} pour la famille des réseaux centrifuges de racine P_0.*

On remarquera que l'ordre $0 > d > c > b > a$ rend les deux réseaux de la preuve du théorème 6.4.1.1 centrifuges.

Preuve Fixons un tel réseau G, et considérons l'algorithme P associé. Au plus un message est échangé sur chaque canal. Tous les calculs de P sont donc finis. Remarquons que les gardes de P_u sont mutuellement exclusives. D'autres part, les propriétés suivantes sont invariantes.

1) $exchanged_u[v] = exchanged_v[u]$
2) Si $data_u$ est différent de *null* alors $data_u = d$.
3) Si $exchanged_u[w]$ et $v > w$ alors $exchanged_u[v]$.

Montrons que tous les calculs de P terminent proprement. Soit C un calcul maximal où P ne termine pas. Au moins un processus est bloqué dans l'état final de C. Soit P_u le processus bloqué de nom maximal. P_u ne peut être bloqué que devant une communication disons avec P_v. Donc $exchanged_u[v]$ est faux. Par l'invariant 1, $exchanged_v[u]$ aussi. Donc, dans l'état final de C, au moins une garde de P_v est fausse et P_v est bloqué aussi. Par maximalité de u, $v < u$.

Par un raisonnement analogue, on montre que, dans l'état final de P_v, toutes les gardes $RTS_v(w)$ et $DTR_v(w)$ sont fausses pour $w > u$. Donc P_v est bloqué devant une communication avec P_u. Puisque $v < u$, $RTS_u(v)$ et $DTR_v(u)$ sont vrais alors que $DTR_u(v)$ et $RTS_v(u)$ sont faux. Cette communication ne peut donc être qu'une émission de P_u vers P_v, et ce à la fois dans P_u et P_v. Une communication est donc possible dans cet état, ce qui contredit la maximalité de C.

Montrons que dans l'état final de C, $data_u = d$ pour tout processus P_u. Considérons la propriété $R(u)$ suivante

Si $exchanged_u[v]$ *pour tous les voisins* P_v *de* P_u *avec* $v > u$, *alors* $data_u = d$.

$R(0)$ est trivialement satisfaite. Considérons un canal $u \longrightarrow w$, avec $w < u$, et supposons que $R(u)$ soit satisfaite. Puisque C termine proprement et $w < u$, P_u envoie un message $new(data_u)$ à P_w. Il ne peut l'envoyer qu'après avoir échangé un message avec tous ses voisins P_v, $v > u$. Donc, puisque $R(u)$ est satisfaite, $data_u = d$ par l'invariant 1. Donc P_w reçoit la donnée $x \neq null$ de P_u, et donc $data_w = d$ après cette réception. Donc $R(w)$ est satisfaite aussi. Puisque G est *centrifuge* par hypothèse, $R(u)$ est vraie pour tout processus P_u, et $data_u = d$ dans l'état final de C pour tout u. ∎

6.5. Multi-diffusion

Les sections précédentes nous ont permis d'étudier le problème de la diffusion d'une valeur dans un réseau à partir d'un processus initiateur. Un problème étroitement lié est celui de la multi-diffusion, où chaque processus possède initialement une donnée, et souhaite la diffuser dans le réseau. L'objectif est que tous les processus acquièrent l'ensemble de toutes les données initialement disponibles. Nous supposons donc dans la suite que chaque processus P_u possède une variable set_u destinée à contenir l'ensemble des données disponibles. Ces données des divers processus sont distinguées par les noms de ces processus. Elles sont de la forme (v, d_v).

Définition 6.5.1. *Soit G un réseau et P un système réparti construit sur G. P est un algorithme de multi-diffusion pour G si*

1) tous les calculs de P terminent proprement;

2) si, dans l'état initial, $set_u = \{(u, d_u)\}$ pour chaque processus P_u, alors dans l'état final $set_u = \{(v, d_v) \mid v \in G\}$ pour tout u.

Il est clair qu'un algorithme de multi-diffusion donne trivialement naissance à un algorithme de diffusion. Les théorèmes 6.1.1 et 6.1.2 s'étendent donc à ce cas. Cependant, contrairement au cas de la diffusion, l'utilisation des gardes d'émission est cruciale pour obtenir un algorithme symétrique.

Théorème 6.5.1. *Il n'existe pas d'algorithme de multi-diffusion symétrique en CSP_{in} dans un anneau.*

Preuve Soit G un anneau, et P un algorithme de multi-diffusion pour G. Soit C un calcul maximal de P. Il est facile de montrer, par un argument analogue à la preuve du théorème 6.1.1, qu'au moins un message est échangé dans C. Donc, pour au moins l'un des processus P_u, P_u peut évoluer de telle manière que sa première communication soit une émission. Par symétrie, c'est le cas de tous les processus de l'anneau. Par restriction et fusion, on obtient un calcul C' de P tel que, dans l'état final de C' tous les processus soient en attente d'émission. Une telle situation en CSP_{in} est nécessairement un blocage général. ∎

Pour obtenir un algorithme de multi-diffusion, il suffit de modifier légèrement les règles de la diffusion par front d'onde d'écrites en section 6.5.1. La condition initiale devient donc la suivante.

Règles du front d'onde (suite)

1') Initialement, chaque processus P_u connaît sa donnée (u, d_u).

Si le réseau est fortement connexe, alors tous les processus auront appris toutes les données disponibles au bout d'un temps fini. Cependant, une implantation directe de ces règles ne suffit cette fois pas à résoudre notre problème. En effet, un processeur n'a maintenant aucun moyen de détecter qu'il a effectivement reçu toutes les données dans la mesure où il ne connaît pas le nombre des processeurs disponibles. Pour reprendre le vocabulaire de [Lehmann 84], on arrive alors à un point où tous les processeurs ont atteint leur état stable, mais aucun ne sait qu'il l'a atteint. Cette situation est en fait très courante dans le dommaine de l'algorithmique répartie. Elle est appelée *terminaison répartie*, pour souligner que, mis à part le fait qu'aucun processus n'a pu centraliser assez d'information pour le détecter, l'état final souhaité est effectivement atteint. Il s'agit donc d'une forme très particulière de blocage, puisque le blocage ne se produit que parce que l'état souhaité a été atteint. Pour obtenir un algorithme de multi-diffusion, il nous faut forcer la terminaison propre de l'algorithme. le chapitre 8 est consacré à l'étude d'une méthode générale. Mais cette méthode utilise elle-même un algorithme de multi-diffusion! Il nous faut donc procèder de manière directe.

L'idée est d'ajouter des informations aux données diffusées. Ces informations ne peuvent être que locales, puisque nous recherchons un algorithme modulaire. Nous considérons donc des données de la forme suivante.

$$(u, d_u, In(u), Out(u))$$

où $In(u)$ et $Out(u)$ sont les ensembles de voisins entrants et sortants de P_u. Sur réception d'un tel message un processus apprend non seulement que la donnée de P_u est d_u, mais aussi qu'il ne doit terminer qu'après avoir appris les valeurs de P_v, pour chaque $v \in (In(u) \cup Out(u))$.

La propriété suivante montre que, réciproquement, la satisfaction de toutes ces contraintes garantit que le processus connaît les données de tous les processeurs du réseau. Définissons le prédicat $COMPLETE(set)$ par

$$COMPLETE(set) = \forall u, v \quad [(u, \_, In, \_) \in set \wedge v \in IN \Longrightarrow (v, \_, \_, \_) \in set]$$

En d'autres termes, $COMPLETE(set)$ est satisfait si à chaque fois que *set* contient la donnée d'un processus, il contient aussi les données de tous ses voisins entrants dans le réseau.

Propriété 6.5.1.. *Supposons le réseau fortement connexe et set non vide. Alors* $COMPLETE(set)$ *est satisfait si et seulement si set contient les données de tous les processeurs du réseau.*

Preuve Considérons la propriété $R(u)$ exprimant que $(u, \_, \_, \_)$ appartient à *set*. Supposons $R(u)$ et considérons un voisin entrant v de u. Alors, puisque $COMPLETE(set)$ est satisfait, $R(v)$. Puisque *set* est non vide, $R(u)$ est satisfaite pour au moins un processeur P_u. Par la remarque précédente, elle est satisfaite pour tous les processeurs dans la composante fortement connexe de P_u, et donc par tous les processeurs du réseau. ∎

Grâce à cette condition, nous pouvons décrire un algorithme de multi-diffusion en $CSP_{i/o}$. Chaque processus P_u utilise un tableau booléen $forwarded[Out]$. $forwarded_u[v]$ est vrai si P_u a envoyé à son voisin sortant P_v le contenu courant de sa variable *set*.Il est remis à faux lors de la reception de toute nouvelle donnée. Un tableau $completed[In]$ est utilisé pour enregister qu'un ensemble maximal de données a été reçu des voisins entrants. Un processus termine lorsque

1) son ensemble de données est maximal: $COMPLETE(set_u)$;
2) cet ensemble a été envoyé à tous ses voisins sortants: $\bigwedge_{out} forwarded[out]$;
3) un tel ensemble a été reçu de chacun de ses voisins entrants: $\bigwedge_{in} completed[in]$.

Le texte de l'algorithme se trouve en figure 6.5.1.

Théorème 6.5.2. *Il existe un algorithme de multi-diffusion modulaire et symétrique en* $CSP_{i/o}$ *pour les réseaux fortement connexes.*

Preuve Soit G un réseau fortement connexe et P le système associé. Montrons que tous les calculs sont finis. Les valeurs envoyées sur chaque canal sont strictement croissantes et bornées. Au plus $N * E$ messages

```
P_u :: [ RESET_u
        *[ ▌ ¬completed[in]; P_in?info(new_set)
           in
           -> old_set:= new_set; set:= set ∪ new_set;
              if set ≠ old_set then forwarded:= false end;
              if COMPLETE(new_set) then completed[in]:= true end
           ▌ ¬forwarded[out]; P_out!info(set)
           out
           -> forwarded[out]:= true
        ]]
```

avec $RESET_u$ qui vaut

set:= $\{(u, d_u, In, Out)\}$; $completed$:= **false**; $forwarded$:= **false**;

Figure 6.5.1.

sont envoyés dans chaque calcul, où N est le nombre de processeurs et E le nombre de canaux de G. Chaque itération inclut l'échange d'un message au moins.

Soit $u \longrightarrow v$ un canal du réseau. Supposons que P_u et P_v soient devant leur boucle externe ou après celle-ci (dans le cas où ils sont terminés) et supposons que $completed_v[u]$ soit vrai. Alors, dans P_u, set_u a pris à un certain moment une valeur telle que $COMPLETE(set_u)$. Par la propriété 6.5.1, cette valeur est maximale, et set_u n'a pas pu être modifiée (incrémentée) par la suite. Donc $forwarded_u$ n'a pas été remis à *false* depuis, et $forwarded_u[v]$ vaut donc toujours *true*.

Soit C un calcul maximal. Nous avons montré qu'il est fini. Supposons que P_u ne termine pas dans C. Alors il est bloqué dans l'état final de C. Le contrôle dans P_u ne peut être que devant sa boucle externe. Supposons que $forwarded_u[v]$ soit faux. Par maximalité, $completed_v[u]$ est vrai et nous obtenons une contradiction. Donc $forwarded_u[v]$ est vrai.

Ceci est aussi satisfait si P_u est terminé. Donc, $forwarded_u[v]$ est vrai pour tout canal $u \longrightarrow v$ de G dans l'état final de C. Donc, dans l'état final de C, $set_u \subseteq set_v$ pour tout canal $u \longrightarrow v$ du réseau G. Comme G est fortement connexe, et que $(u, d_u, \_, \_) \in set_u$ pour tout u, on en déduit que $COMPLETE(set_u)$ est vrai pour tout u. Donc $completed_v[u]$ est vrai et tous les processus sont en fait terminés. ▌

On remarquera qu'il n'est pas nécessaires de tester explicitement $COMPLETE(set_u)$. D'après la preuve, si $completed_u[v]$ est vrai pour un voisin entrant P_v alors cette propriété est implicitement satisfaite.

On peut se demander comment cet algorithme se comporte dans le cas d'un réseau *non* fortement connexe. On peut vérifier que le système termine proprement. A la terminaison, chaque processus P_u possède dans sa variable set_u l'ensemble des données des processeurs P_v tels qu'il existe un chemin de v vers u dans G (le cône de sommet u dans G). Les processus peuvent de plus détecter le manque de forte connexité testant le prédicat $PENDING(set)$

$$PENDING(set) = \exists u, v \quad [(u, \_, \_, Out) \in set \wedge (v \in Out) \wedge (v, \_, \_, \_) \notin set]$$

En fait, il est possible dans ce cas d'utiliser les techniques vues en section 6.2.3.4 pour obtenir un algorithme de multi-diffusion pour les réseaux faiblement connexes. Notons que ceci implique un codage (binaire par exemple) des identificateurs de processus. Implicitement, ceci suppose donc de considérer que l'ensemble $Name$ est en fait l'ensemble des entiers naturels. D'autre part, la symétrie est perdue, puisque les types des messages échangés dépendent, via ce codage, du nom des processus.

6.6. Conclusions

Nous ne nous étendrons pas ici sur les conclusions liées au pouvoir expressif des divers dialectes de CSP qui seront développées en section 7.5.1. Au niveau algorithmique, l'aspect le plus intéressant est la notion d' **apprentissage**. Le problème est de permettre à un ensemble de processus d'apprendre, dynamiquement et coopérativement, la structure topologique du réseau dans lequel ils sont plongés. Un algorithme d'apprentissage est obtenu à partir de l'algorithme de multi-diffusion décrit en section 6.5 en considérant que la donnée du processus P_u est précisément son environnement local $(u, In(u), Out(u))$. A la terminaison, chaque processus possède une image set_u du cône dont il est le sommet. $PENDING(set_u)$ est satisfait si et seulement si cette image n'est pas une composante fortement connexe de G.

Théorème 6.6.1. *Il existe un algorithme modulaire et (fonctionnellement) symétrique d'apprentissage en $CSP_{i/o}$ pour les réseaux fortement connexes. Il existe un algorithme modulaire et symétrique en $CSP_{i/o}$ pour tester la forte connexité des réseaux faiblement connexes.*

7. Algorithmes symétriques d'élection

Au chapitre 4, nous avons discuté la notion de symétrie dans le cadre des systèmes répartis en CSP. Nous avons montré l'insuffisance d'une définition syntaxique de la symétrie. Nous avons proposé une définition sémantique qui semble rendre compte de manière plus exacte de l'intuition sous-jacente. Le but de ce chapitre est d'évaluer la pertinence de notre définition sémantique à propos d'un problème précis, celui de l'élection. Nous allons montrer que la notion de symétrie permet de définir des conditions fines pour l'existence ou la non existence de solutions à ce problème. De plus, il se trouve que ces conditions sont étroitement liées à l'utilisation par le langage CSP des gardes de communication, émission ou réception. Ces conditions fournissent donc indirectement une évaluation précise du pouvoir d'expression de cette facilité. Nous montrerons en particulier que la notion de garde de communication doit être considérée comme primitive dans la sémantique de CSP, contrairement par exemple à la convention de terminaison répartie (*cf.* [AF 84]). Le matériel de ce chapitre a déjà été largement présenté en [Bougé 86].

7.1. Historique

Le problème de l'élection d'un coordinateur ("leader") au sein d'un réseau de processeurs a été étudié par de nombreux auteurs dans divers cadres théoriques.

Il semble que le problème ait été identifié pour la première fois par Le Lann [LeLann 77]. En fait, l'élection était alors considérée, non pour elle-même, mais comme une partie d'un protocole d'exclusion mutuelle résistant aux pannes. Il fallait en effet assurer que, lors d'une reconfiguration dynamique du système à la suite d'une panne, un et un seul jeton de contrôle soit créé. Le problème était alors compliqué par la possibilité de pannes pendant l'élection elle-même. L'algorithme d'élection proposé par Le Lann est fondamentalement du type de celui décrit au chapitre 4, figure 4.1. Il n'est pas symétrique puisque c'est toujours le processeur de nom maximal qui est finalement élu.

Une approche générale des problèmes d'élection au sein des réseaux de processeurs est proposée par Garcia-Molina [Garcia 82]. Comme pour Le Lann, l'élection y est vue principalement comme une technique de reconfiguration dynamique à la suite d'une panne. Différents types d'élections sont envisagés suivant le type de panne. Le mécanisme utilisé est essentiellement la notion de priorité statique. Les processeurs de priorité haute imposent leur choix à ceux de priorité plus basse. De plus, les processeurs de priorité basse observent une période de contention plus longue que ceux de priorité haute. Le cadre dans lequel ce travail se situe est donc très éloigné de celui proposé par Hoare dans CSP.

De nombreux travaux ont été consacrés à la recherche d'algorithmes d'élection efficaces pour des anneaux de processeurs. Le choix d'une telle topologie tient à la remarque suivante. Sur un anneau, il est naturel de visiter tous les sommets successivement en ne passant qu'une fois par chacun d'eux (chemin hamiltonien). De plus, si l'anneau est dirigé, chaque processeurs peut déterminer localement son successeur dans ce chemin privilégié. C'est donc une propriété de modularité. Typiquement, la diffusion de l'information se fera par un jeton parcourant ce chemin. Le plus souvent, l'efficacité de l'algorithme est mesurée par le nombre de messages échangés pendant une élection. Les algorithmes habituellement considérés dérivent de celui proposé par Le Lann. Ils ne sont donc pas symétriques au sens où nous l'entendons.

L'efficacité des algorithmes proposés dépend crucialement du cadre dans lequel on se place. Le cas des systèmes synchrones a été étudié par Gafni, Afek et Kleinrock ([GAK 84]) dans le cas des réseaux complets bidirectionnels. Leurs hypothèses impliquent entre autres la modularité: initialement, un processeur ne connaît que son nom (unique) et les canaux adjacents sont pour lui indistingables. Les processeurs ont accès à une horloge globale permettant de définir des phases ("rounds"). Ces auteurs décrivent un algorithme optimal en temps et en nombre des messages échangés. Frederickson et Nancy Lynch ([FL 84]) étudient l'influence du degré de synchronisation entre les processus. L'idée est d'utiliser le nom des processeurs (des entiers naturels distincts) pour moduler la vitesse de leurs messages le long de l'anneau. On parvient ainsi à ce que les processeurs de nom peu élevé soient très vite éliminés de la compétition. Contrairement aux

algorithmes classiquement étudiés, non seulement l'ordre relatif des noms mais bien leur valeur absolue est donc utilisée! L'accès à une horloge globale permet donc de diminuer strictement le nombre de messages.

Cependant, c'est le cas asynchrone qui a été le plus étudié. Une synthèse des différents résultats est donnée par Vitanyi ([Vitanyi 84]). En fait, ceux-ci dépendent crucialement du type d'anneau qui est considéré. Les anneaux bidirectionnels avec sens de l'orientation permettent des élections plus efficaces que les anneaux unidirectionnels. En effet, les jetons signés de l'algorithme de Le Lann mettent en moyenne moins de temps pour atteindre un processeur de nom plus élevé. Vitanyi propose un algorithme proche de celui de Le Lann en supposant un asynchronisme borné (Les temps locaux sont archimédien deux à deux). Ceci permet, comme dans le cas synchrone, d'utiliser le nom d'un processus dans le calcul de périodes de contention privilégiant les processus de noms élevés. D'autres résultats peuvent être trouvés dans [KRS 83] ou [GN 84] pour les anneaux bidirectionnels. Enfin, le cas de réseaux fortement connexes arbitraires est étudié par Gafni et Afnek ([GA 84]).

Le problème de l'élection est en fait étroitement relié à de nombreux autres problèmes classiques concernant les systèmes répartis. On peut citer par exemple la construction dynamique d'arbres recouvrants (la racine peut être alors considérée comme élue); la recherche d'une valeur maximale (on l'applique au cas où les valeurs sont les noms eux-mêmes des processus); le problème de l'exclusion mutuelle (le processus ayant accès en premier à la section critique est élu) *etc.* D'autre part les algorithmes de consensus byzantin permettent de concevoir aisément des algorithmes d'élection tolérants aux pannes ([Merritt 84]). Le problème de l'élection est donc au coeur de l'algorithmique des systèmes répartis. C'est pourquoi l'étude de ce problème et la comparaison des résultats obtenus dans divers cadres théoriques revêt une grande importance.

Dans son travail, Dana Angluin ([Angluin 80]) étudie le problème de la détermination d'un centre dans un réseau d'automates communicants par des canaux synchrones. Le cadre dans lequel elle se place est très proche de $CSP_{i/o}$ où l'on admet des gardes d'émission et de réception. De fait, elle obtient des résultats positifs et négatifs qui sont des cas particuliers des théorèmes généraux que nous obtenons. Notons que, par nature, les systèmes répartis qu'elle considère sont symétriques et modulaires.

Johnson et Schneider ([JS 85]) considèrent l'existence des solutions symétriques (en fait, "similaires") au problème des philosophes de Dijkstra. Ils se placent dans le cadre des communications par partage de variables. Ils obtiennent des résultats positifs et négatifs d'existence. Ces résultats ne semblent cependant pas être comparables directement aux nôtres du fait de la différence entre les langages (variables partagées d'une part, messages synchrones d'autre part).

7.2. Systèmes d'élection symétriques

Nous définissons maintenant le problème de l'élection symétrique en CSP. Soit G un réseau et P un système réparti construit sur G. Nous supposons que chaque processus P_u de P déclare une variable locale *decision* de type *Name*.

Définition 7.2.1. *Soit P un système réparti construit sur un graphe G. P est un système d'élection pour G si*

1) tout calcul C de P termine proprement;

2) à la terminaison, toutes les variables decision contiennent la même valeur v qui est le nom d'un des processus P_v de P.

On dira alors que cette valeur v est le nom du processus élu. On note $v = decision(C)$ où C est un calcul maximal. On dira que P est un système d'élection symétrique pour G si P est symétrique et si la fonction de décision respecte la symétrie.

Définition 7.2.2. *Soit P un système réparti construit sur un réseau G, soit Σ_G son groupe d'automorphismes. P est un système d'élection symétrique pour G si P est un système d'élection pour G, s'il est symétrique, et si les applications S_σ de la définition 4.3.1 vérifient de plus*

6) pour tout calcul C maximal,

$$decision(S_\sigma(C)) = \sigma(decision(C))$$

Le système P est donc en fait fonctionnellement symétrique (*cf.* definition 5.2.2.1), pour une action E_σ sur l'environnement telle que

$$E_\sigma(env)(decision_u) = \sigma(env(decision_u))$$

Un système d'élection symétrique est donc précisément un système d'élection qui est fonctionnellement symétrique pour cette action. On remarquera que la condition 6 ci-dessus est consistante. Si C est un calcul terminé, il est maximal. Donc $S_\sigma(C)$ est aussi maximal et, par définition des systèmes d'élection, les calculs maximaux ne peuvent être que proprement terminés. Remarquons tout de suite que cette définition est sujette aux mêmes remarques que celle de la symétrie. En particulier, si G est un graphe ayant l'identité pour seul automorphisme, tout système d'élection est un système d'élection symétrique. Comme pour la symétrie, on définira les systèmes d'élection Σ-symétriques, où Σ est un sous-groupe de Σ_G, en ne considérant que les automorphismes de Σ dans la définition 7.2.2. Un système d'élection symétrique est donc un système d'élection Σ_G-symétrique. Si $\Sigma_0 \subseteq \Sigma_1$, tout système d'élection Σ_1-symétrique est un système d'élection Σ_0-symétrique.

Etant donné un réseau G, le problème de l'élection symétrique pour G peut être formulé comme suit.

Existe-t-il un système réparti P qui soit un système d'élection symétrique pour G?

On distinguera pour répondre à cette question les différents dialectes CSP_{no}, CSP_{in} et $CSP_{i/o}$ présentés au chapitre 1.

Soit $\mathcal{G}$ une famille de réseaux, et $\mathcal{P}$ une famille de systèmes répartis bâtis sur ces réseaux. Si chaque système P de la famille est un système d'élection relativement à son réseau, on dira que $\mathcal{P}$ est un algorithme d'élection symétrique pour $\mathcal{G}$. Si la famille $\mathcal{P}$ est modulaire, on parlera d'un algorithme d'élection modulaire et symétrique. Les notions de théorie des graphes utilisées dans ce chapitre sont présentées en section 2.1.

Nous commencerons cette étude par un résultat de prolongement. Ce résultat exprime que pour problème de l'élection symétrique, l'étude des "grands" graphes se ramène à l'étude des "petits" graphes homogènes.

Théorème 7.2.1. *Soit G un graphe fortement connexe. Soit Σ_G son groupe d'automorphismes. Soit G' un sous-graphe de G invariant par Σ_G. Si G' admet un système d'élection symétrique en $CSP_{i/o}$ alors G aussi.*

Preuve Soit P' un système d'élection symétrique pour G'. Noter que P' est $\Sigma_{G'}$-symétrique par construction, et que (un quotient de) Σ_G est isomorphe à un sous-groupe de $\Sigma_{G'}$. Nous construisons un système d'élection symétrique P pour G en composant trois phases, chacune d'elles étant Σ_G-symétrique (en fait, fonctionnellement symétriques au sens du chapitre 5). Les théorèmes généraux du chapitre 5 montrent que leur composition est symétrique.

Phase 1

Les processus composant G' exécutent le système d'élection P' tandis que les autres restent inactifs. Cette phase est Σ_G-symétrique puisque (un quotient de) Σ_G se plonge dans $\Sigma_{G'}$. Le nom du processus élu se trouve dans la variable $decision'$.

Phase 2

Chaque processus P_u composant G' exécutent $decision_u := decision'_u$.

Phase 3

Les processeurs de G exécutent l'algorithme de diffusion modulaire et symétrique de la section 6.2.1. Le processeur P_u tel que $decision_u = u$ joue le rôle de P_0.

Remarquons que les phases 2 et 3 sont en fait modulaires pour la famille des graphes fortement connexes. On en déduit le corollaire suivant.

Corollaire 7.2.1. *Soit $\mathcal{G}$ une famille de graphes fortement connexes. Pour chaque graphe G de $\mathcal{G}$, soit G' un sous-graphe invariant, et soit $\mathcal{G}'$ la famille ainsi formée. Tout algorithme d'élection modulaire et symétrique en $CSP_{i/o}$ pour $\mathcal{G}'$ donne naissance à un algorithme d'élection symétrique et modulaire en $CSP_{i/o}$ pour $\mathcal{G}$.*

De manière plus générale, selon les résultats du chapitre 6, ce théorème s'étend au cas de CSP_{in} si le réseau est bidirectionnel. Le théorème ci-dessus montre qu'un système d'élection symétrique pour un sous-graphe invariant s'étend au graphe entier. Par exemple, pour le graphe présenté en figure 7.2.1.

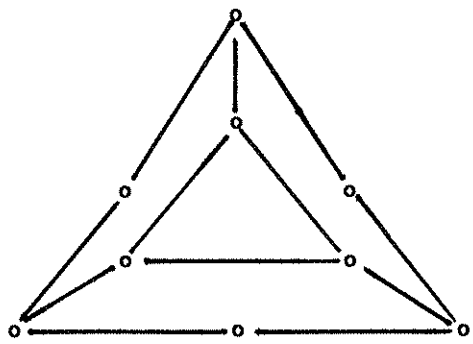


Figure 7.2.1.

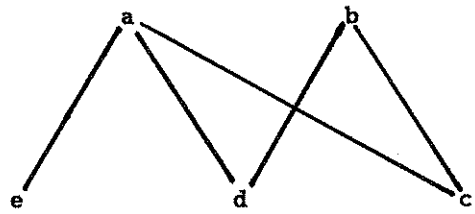


Figure 7.2.2.

Un système d'élection symétrique pour le triangle interne détermine un système d'élection symétrique pour le graphe entier. La réciproque est fausse en général, comme le montre le théorème suivant.

Théorème 7.2.2. *Un réseau non (faiblement) connexe homogène n'admet aucun système d'élection symétrique.*

Preuve Soit G un tel graphe. Partageons G en deux sous-graphes disjoints G_1 et G_2. Soient u_1 un sommet de G_1, et u_2 un sommet de G_2. Soit P un système d'élection symétrique pour G. Soit C un calcul maximal de P. C aboutit à l'élection d'un sommet u. Par homogénéité, on peut trouver un automorphisme σ_i de G tel que $\sigma_i(u) = u_i$, $i = 1, 2$. Soit $C_i = S_{\sigma_i}(C)$, $i = 1, 2$. Comme G_1 et G_2 sont disjoints, leurs processeurs ne peuvent communiquer dans les calculs de P. Les transitions effectuées par ces processeurs sont donc concurrentes. Par commutation et restriction, on peut trouver un calcul C'_1 tel que pour tout sommet u de G_1, P_u se comporte dans C'_1 comme dans C, alors que pour tout sommet u de G_2, P_u est inactif dans C_1. On peut faire de même avec C_2. Les calculs C'_1, et C'_2 peuvent être fusionnés en un calcul C'. Dans l'état final de C', $decision_{u_i} = u_i$ pour $i = 1, 2$. Comme $u_1 \neq u_2$, ceci contredit la définition des systèmes d'élection. ∎

Un contrexemple est alors obtenu en appliquant ce théorème à un graphe dont tous les sous-graphes invariants non triviaux sont non connexes, comme celui présenté en figure 7.2.2.

Un autre point important est celui de la généralité de l'élection. La définition des système d'élection n'impose pas que tout processeur puisse être élu. En fait, le théorème suivant montre que cette restriction n'est pas nécessaire. Nous dirons qu'un système d'élection P construit sur un réseau G est **homogène** si tout sommet de G peut être élu par P.

Théorème 7.2.3. *Soit G un graphe fortement connexe. Si G admet un système d'élection symétrique en $CSP_{i/o}$ alors il admet un système d'élection symétrique homogène en $CSP_{i/o}$.*

Preuve Soit P un système d'élection symétrique pour G. On construit P' par la composition répartie de trois phases symétriques.

Phase 1

Tous les processeurs exécutent le système d'élection symétrique P.

Phase 2

Chaque processeur P_u teste si $decision_u = u$. Si c'est le cas, P_u choisit de manière non déterministe un sommet v de G. Cette phase est atomique. Si v est choisi dans un calcul C, $\sigma(v)$ est choisi dans $S_\sigma(C)$.

Phase 3

Tous les processus exécutent un algorithme de diffusion symétrique. Le processeur (unique) P_u tel que $decision_u = u$ est l'initiateur et diffuse la valeur v. A la terminaison, les processeurs affectent la valeur diffusée à la variable *decision*. ∎

Comme précédemment, nous obtenons des variantes à ce corollaire en considérant les différents algorithmes de diffusion vus au chapitre 6. En particulier, il est possible de préserver la modularité en $CSP_{i/o}$. Si le réseau est bidirectionnel, on obtient un résultat analogue en CSP_{in}.

7.3. Election symétrique en CSP_{no} et CSP_{in}

Nous considérons dans cette section le problème de l'élection symétrique en CSP_{no} et CSP_{in}.

Etant donné un réseau G, existe-t-il un système d'élection symétrique P construit sur G en CSP_{no} ou CSP_{in}?

De manière plus générale, on peut formuler la question ainsi.

Etant donné une famille $\mathcal{G}$ de réseaux, peut-on trouver une famille $\mathcal{P}$ modulaire de tels systèmes d'élection symétrique?

Rappelons que le problème de l'homogénéité des systèmes obtenus est pris en compte par les résultats généraux de la section précédente.

7.3.1. Résultat positif

Nous commençons par des résultats d'existence très simples, conséquences directes des théorèmes généraux de la section 2. Un premier cas trivial se présente lorsqu'un graphe n'a qu'un seul automorphisme, l'identité. Un exemple de tel graphe est présenté en figure 4.2.1. Pour un tel graphe, tout système réparti est symétrique.

Théorème 7.3.1.1. *Soit G un réseau centrifuge de racine P_0 dont le groupe d'automorphismes n'a qu'un élément. Il existe un système d'élection symétrique en CSP_{no} pour G.*

Preuve P_0 choisit l'un des processeurs de G, et diffuse cette donnée grâce à l'algorithme de diffusion symétrique en CSP_{no} de la section 6.3.3. Chaque processus exécute simplement

$$decision_u := d_u .$$

ce qui constitue trivialement une phase (fonctionnellement) symétrique, puisque qu'aucune communication n'a lieu. On applique alors les resultats du chapitre 5. ∎

Ce résultat est évidemment très faible, et nous n'avons pu en trouver de meilleur. Ceci s'explique par le compromis nécessaire entre la symétrie et les gardes de communication.

Si l'on admet des gardes de reception, l'on peut affaiblir légèrement les conditions. Rappelons qu'un graphe G a un point fixe si pour tout automorphisme σ de G, $\sigma(u) = u$.

Théorème 7.3.1.2. *Soit G un graphe bidirectionnel connexe ayant un point fixe. Alors il existe un système d'élection symétrique homogène pour G en CSP_{in}.*

Preuve Soit G' le sous-graphe de G constitué par l'un de ses points fixes u. G' est invariant et admet trivialement un système d'élection symétrique en CSP_{in}. On applique alors le théorème 7.2.1. légèrement modifié. P_u choisit de manière non deterministe l'un des sommets v de G et diffuse cette donnée au lieu de son propre nom. Remarquons que l'on perd en general la modularité, puisque nous ne connaissons pas de système d'apprentissage modulaire et symétrique en CSP_{in}. ∎

Les conditions énoncées par les théorèmes ci-dessus sont loin d'être nécessaires comme le montre l'exemple suivant. Considérons le graphe G présenté en figure 7.2.2 (graphe bipartite complet (2,3)).

Propriété 7.3.1.1. *Le graphe G ci-dessus admet un système électoral symétrique en CSP_{in}.*

Preuve Un système électoral symétrique pour G peut être obtenu par composition des phases (fonctionnellement) symétriques suivantes.

Phase 1

P_a envoie un jeton blanc à P_e, P_d ou P_c et un jeton noir à chacun des deux autres processeurs. P_b fait de même. P_e, P_d et P_c attendent chacun un jeton sur chacun de leurs canaux (par une répétition non déterminisme avec gardes de réception).

Phase 2

P_e, P_d et P_c envoient à P_a et P_b les deux jetons qu'ils ont reçus. P_a et P_b attendent deux jetons sur chacun de leurs canaux.

Phase 3

P_a et P_b décident du résultat de l'élection selon les critères suivants. Si l'un des processus P_e, P_d ou P_c a reçu deux jetons blancs, il est élu. Sinon, celui qui a reçu deux jetons noirs est élu. ∎

7.3.2. Premier résultat négatif

Comme la faiblesse des résultats positifs obtenus ci-dessus le laisse pressentir, nous allons montrer qu'une large classe de réseaux n'admet pas de système électoral symétrique en CSP_{in}. Nous avons déjà noté à plusieurs reprises que la symétrie et l'absence de gardes d'émission sont deux paramètres contradictoires. Ce fait a été souligné de nombreuses fois dans divers travaux; on se reportera par exemple à [Hoare 78] et [Bernstein 80] dans le cas de CSP, ou à [LR 81] pour une constatation analogue dans le cas des communications par variables partagées. La remarque fondamentale est qu'un calcul d'un système P en CSP_{in} dans un état où il existe un sous-ensemble Q de processus de P qui soient tous en attente d'émission vers un processus de Q ne peut terminer proprement.

Dans cette section, nous montrons unn premier type de résultat négatif tout-à-fait analogue à celui présenté en [LR 81]. La section suivante présentera une généralisation de ce résultat, fondée sur une méthode de preuve originale. Nous commençons par un lemme fondamental pour ce type de résultat.

Lemme 7.3.2.1. *Soit G un réseau sans point fixe, et soit P un système électoral pour G. Tout calcul maximal de P contient au moins une communication.*

Preuve Soit C un calcul terminé de P ne contenant aucune communication. Puisque P est un système électoral symétrique, un processus, disons P_u, est élu à la terminaison de P. Par hypothèse, il existe un automorphisme σ de G tel que $\sigma(u) = v$ avec $u \neq v$. Considérons le calcul $\sigma(C)$ (plus exactement, $S_\sigma(C)$). Ce calcul aboutit à l'élection de $\sigma(u) = v$. Comme C, il ne contient aucune communication.

Considérons le calcul (partiel) C_1 défini par commutation et restriction à partir de C comme suit. Dans C_1, P_u se comporte comme dans C, alors que tous les autres processus restent inactifs. De même, considérons le calcul C_2 obtenu comme suit. Dans C_2, P_v se comporte comme dans $\sigma(C)$ alors que tous les autres processus restent inactifs. Soit C_3 la fusion de C_1 et de C_2. Dans l'état final de C_3, les processus P_u et P_v sont terminés. Les variables $decision_u$ et $decision_v$ ont respectivement les valeurs u et v. Par définition des systèmes électoraux symétriques, C_3 se prolonge en calcul terminé C_4. Dans l'état final de C_4, les variables $decision_u$ et $decision_v$ ont encore les valeurs ci-dessus. Comme $u \neq v$, ceci contredit la definition d'un système électoral symétrique. ∎

Nous pouvons maintenant appliquer ce lemme pour montrer le théorème suivant. Rappelons qu'un graphe est **homogène** si pour tous sommets u et v il existe un automorphisme σ tel que $\sigma(u) = v$. Un tel graphe vérifie donc l'hypothèse du lemme 7.3.2.1.

Théorème 7.3.2.1. *Un réseau homogène non trivial n'admet aucun système électoral symétrique en* CSP_{in}.

Preuve Soit G un tel réseau et soit P un système électoral symétrique pour G. Soit C_0 un calcul maximal de P. Par définition, C_0 termine proprement. Par le lemme 7.3.2.1, il contient au moins une communication. Considérons alors la *première* communication de C_0. Disons que c'est une émission de P_a vers P_b. Par commutation et restriction, il existe un calcul partiel C de P tel que P_a et P_b se comportent comme dans C_0 jusqu'à leur première communication inclue, alors que tous les autres processus restent inactifs. Maintenant, pour tout sommet u, il existe un automorphisme σ_u tel que $\sigma_u(a) = u$, puisque G est un réseau homogène. $\sigma_u(C)$ est donc un calcul de P où P_u et $P_{\sigma_u(b)}$ établissent exactement une communication, alors que tous les autres processus restent inactifs.

Considérons le calcul C_u obtenu par commutation et restriction à partir de $\sigma_u(C)$ de la manière suivante. Dans C_u, P_u progresse jusqu'à sa première communication (qui est une émission) *exclue*, alors que *tous* les autres processus restent inactifs. Le calcul C_u ne contient donc aucune communication et seul P_u y est actif. Tous ces calculs peuvent donc être fusionnés en un calcul C_1. Dans l'état final de C_1, le contrôle dans chaque processus est devant une émission. Comme nous l'avons déjà remarqué plus haut, cette situation ne peut se produire dans un système électoral symétrique en CSP_{in}. ∎

Un premier mouvement du lecteur pourrait être de combiner ce théorème avec les remarques suivant le théorème 7.2.1. Il est vrai que tout réseau admet un sous-réseau invariant homogène. Cependant, nous avons déjà remarqué que le réciproque du théorème 7.2.1 est fausse. Un réseau peut admettre un système électoral symétrique alors qu'aucun de ses sous-réseaux invariants n'en admet!

Le théorème ci-dessus montre que les réseaux complets (cliques) non triviaux et les anneaux (uni- ou bidirectionnels) n'admettent pas de système électoral symétrique en CSP_{in}. Le cas de CSP_{no} est encore plus restrictif comme le montre la propriété ci-dessous. Soit G le réseau de la figure 7.2.2.

Propriété 7.3.2.1. *Le réseau G ci-dessus n'admet pas de système électoral symétrique en* CSP_{no}.

Preuve Soit P un système électoral symétrique pour ce réseau en CSP_no. Soit C_0 un calcul maximal de P. Il est facile de vérifier que le lemme 7.3.2.1 s'applique ici. C_0 contient donc au moins une communication. Considérons la *première* communication de C_0. Disons que c'est une communication entre P_a et P_e. En utilisant la méthode du théorème 7.3.2.1, on peut trouver des calculs partiel $C_{u,v}$ de P, $u = a, b$, $v = c, d, e$, avec la propriété suivante. Dans $C_{u,v}$, seul P_u est actif, et il progresse jusqu'à une communication (exclue) avec P_v.

Considérons alors la fusion des calculs $C_{a,e}$, $C_{b,d}$, $C_{e,b}$, $C_{d,a}$, $C_{c,a}$. Dans l'état final de C_1, le contrôle dans chaque processus est devant une commande de communication, et aucune communication ne peut s'établir. Puisque nous sommes en CSP_{no}, ce calcul est donc maximal. Il ne peut être terminé. Contradiction. ∎

7.3.3. Second résultat négatif

Les exemples précédents ont montré la puissance des notions de commutation et de restriction. Les méthodes utilisées sont cependant classiques dans leur principe. Fondamentalement, elles se résument ainsi. Dans un système symétrique, il est impossible d'empêcher les processus de faire tous "la même chose". En présence de communications bloquantes, le blocage est donc inévitable. Ce principe avait déjà été appliqué par [LR 81] et [Burns 81] dans d'autres cadres. Dans le cas qui nous intéresse ici, il est possible d'utiliser d'autres principes plus subtils conduisant à des résultats beaucoup plus puissants. Rappelons qu'un automorphisme σ d'un graphe G est **équilibré** si toutes ses orbites ont même cardinal. En particulier, l'automorphisme trivial Id est équilibré.

Théorème 7.3.3.1. *Un réseau ayant un automorphisme équilibré non trivial n'admet aucun système électoral symétrique en* CSP_{in}.

Preuve Soit G un tel réseau et σ un automorphisme équilibré non trivial de G. Soit m la taille commune des orbites de σ. Nous allons construire une suite croissante de calculs partiels

$$C_0 \prec C_1 \prec C_2 \prec \cdots$$

satisfaisant les conditions suivantes.

1) C_{i+1} contient au moins une communication de plus que C_i.
2) $\sigma^{(0)}(C_i) = C_i$, $\sigma^{(1)}(C_i)$, …, $\sigma^{(m-1)}(C_i)$ sont causalement équivalents.

La condition 1 montre que P admet un calcul infini, limite croissante des calculs partiels C_i, ce qui contredit la définition des systèmes électoraux symétriques. La construction se fait par récurrence sur i. Le cas de base $i = 0$ est trivial. Il suffit de prendre pour C_0 le calcul vide. On a déjà vu que les bijections S_σ conservent la longueur des calculs. L'image du calcul vide est donc le calcul vide. Posons maintenant $C = C_i$ et construisons C_{i+1}. Les lemmes intervenant dans cette construction seront démontrés par la suite.

Lemme 7.3.3.1. *C n'est pas un calcul maximal.*

Soit donc C_0 un calcul maximal prolongeant C.

Lemme 7.3.3.2. *$C_0 \setminus C$ contient au moins une communication.*

On peut donc trouver deux processus P_a et P_b tels que
1) P_a et P_b communiquent dans $C_0 \setminus C$;
2) La première communication de P_a dans $C_0 \setminus C$ est une émission vers P_b;
3) La première communication de P_b dans $C_0 \setminus C$ est une réception de P_a.

Noter qu'il suffit de considérer par exemple la première communication du calcul $C_0 \setminus C$. Soit O_a et O_b les orbites de a et b sous l'action de σ. Elles ont m pour cardinal.

Lemme 7.3.3.3. *O_a et O_b sont des orbites disjointes.*

Considérons alors le calcul $C_{a,b}$, avec

$$C \prec C_{a,b} \prec C_0$$

défini comme suit. Dans $C_{a,b} \setminus C$, P_a et P_b progressent jusqu'à leur première communication incluse, tandis que tous les autres processus restent inactifs. Considérons alors les calculs $\sigma^{(0)}(C_{a,b}) = C_{a,b}$, $\sigma^{(1)}(C_{a,b})$, ..., $\sigma^{(m-1)}(C_{a,b})$. Ces calculs étendent respectivement $\sigma^{(0)}(C)$, $\sigma^{(1)}(C)$, ..., $\sigma^{(m-1)}(C)$. Par hypothèse, ces derniers sont causalement équivalents. Ils ont donc même état final. Dans $\sigma^{(p)}(C_{a,b}) \setminus \sigma^{(p)}(C)$, seuls $P_{\sigma^{(p)}(a)}$ et $P_{\sigma^{(p)}(b)}$ sont actifs. Par le lemme 7.3.3.3, tous ces $\sigma^{(p)}(a)$, $\sigma^{(p)}(b)$, $p = 0, 1, \ldots, m-1$ sont distincts. On peut donc fusionner tous ces calculs au-dessus de C pour obtenir un calcul C' de la forme suivante

$$C; \sigma^{(0)}(C_{a,b}) \setminus \sigma^{(0)}(C); \cdots; \sigma^{(m-1)}(C_{a,b}) \setminus \sigma^{(m-1)}(C)$$

Lemme 7.3.3.4. *C' et $\sigma^{(p)}(C')$, $p = 0, 1, \ldots, m-1$ sont causalement équivalents.*

Le calcul C' est précisément le calcul C_{i+1} cherché. Par définition, il étend C. Il contient $C_{a,b} \setminus C$ qui contient une communication. Le lemme 7.3.3.4 est exactement la seconde propriété de récurrence. ∎

Il nous reste à montrer les quatre lemmes.

Preuve du lemme 7.3.3.1

Si C était maximal, il serait terminé et conduirait à l'élection de P_u. Donc $\sigma(C)$ serait aussi terminé et conduirait à celle de $P_{\sigma(u)}$. Mais C et $\sigma(C)$ sont causalement équivalents. Ils ont donc même état final. Donc $u = \sigma(u)$. L'orbite de u est donc de cardinal 1. Comme σ est équilibré, il est donc trivial. Contradiction. ∎

Preuve du lemme 7.3.3.2

Supposons que $C_0 \setminus C$ ne contienne aucune communication. C_0 conduit à l'élection de u. $C'_0 = \sigma(C_0)$ étend $C' = \sigma(C)$ et conduit à celle de $\sigma(u) \neq u$. Par commutation et restriction, on peut trouver un calcul C_1, $C \prec C_1 \prec C_0$, tel que P_u se comporte dans C_1 comme dans C_0 alors que tous les autres processus sont inactifs dans $C_1 \setminus C$. De même, on peut trouver un calcul C'_1, $C' \prec C'_1 \prec C'_0$, tel que $P_{\sigma(u)}$ se comporte dans C'_1 comme dans C'_0 alors que tous les autres processus restent inactifs dans $C'_1 \setminus C'$.

C et C' sont causalement équivalents, et ont donc même état final. On peut donc fusionner C_1 et C'_1 au-dessus de C en un calcul C_2. Dans l'état final de C_2, P_u et P'_u sont terminés. Leur variable *decision* a respectivement les valeurs u et $\sigma(u)$. Soit C_3 un calcul terminé prolongeant C_2. C'est aussi le cas dans l'état final de C_3, et donc $u = \sigma(u)$. Contradiction. ∎

Preuve du lemme 7.3.3.3

Supposons que O_a et O_b ne soient pas disjointes. Alors, elle sont confondues, et il existe p, $0 < p < m$, tel que $b = \sigma_{(p)}(a)$. Considérons le calcul $C_{a,b}$,

$$C \prec C_{a,b} \prec C_0$$

défini comme suit. Dans $C_{a,b} \setminus C$, les processus P_a et P_b progressent jusqu'à leur première communication incluse, alors que tous les autres processus restent inactifs. Cette communication est une émission de P_a vers P_b.

Considérons alors $\sigma^{(0)}(C_{a,b}) = C_{a,b}$, $\sigma^{(1)}(C_{a,b})$, ..., $\sigma^{(q-1)}(C_{a,b})$, q étant le plus petit entier positif non nul tel que m divise $p*q$ (q estle quotient du ppcm(m,p) par p). Dans $\sigma^{(r)}(C_{a,b}) \setminus C$, seuls $P_{\sigma^{(r)}(a)}$ et $P_{\sigma^{(r)}(b)}$ sont actifs.

Considérons maintenant le calcul $C_a^{(r)}$

$$\sigma^{(r)}(C) \prec C_a^{(r)} \prec \sigma^{(r)}(C_{a,b})$$

otenu par commutation et restriction à partir de $\sigma^{(r)}(C_{a,b})$ pour $r = 0, p, \ldots, (q-1)*p$. Dans $C_a^{(r)}$, seul $P_{\sigma^{(r)}(a)}$ est actif et progresse jusqu'à sa première communication exclue. Cette communication est une émission vers $P_{\sigma^{(r)}(b)}$.

Par choix de q, $\sigma^{(0)}(a) = a$, $\sigma^{(p)}(a) = b$, ..., $\sigma^{(q-1)*p}$ sont disctincts. D'autre part, tous les $\sigma^{(r)}(C)$ ont même état final. On peut donc fusionner au-dessus de C les calculs $C_a^{(0)}$, $C_a^{(p)}$, ..., $C_a^{(q-1)*p}$, ce qui produit un calcul C_1. Dans l'état final de C_1, P_a est devant une commande d'émission vers $P_{\sigma(p)(a)}$, $P_{\sigma(p)(a)}$ vers $P_{\sigma(2*p)(a)}$, *ldots*, $P_{\sigma(q-1)*p(a)}$ vers P_a. Nous avons déjà remarqué qu'un système électoral symétrique en CSP_{in} ne peut passer par un tel état. Contradiction. ∎

Preuve du lemme 7.3.3.4

Posons $C_{a,b} = D$, $\sigma^{(i)}(C) = C^i$, $\sigma^{(i)}(D) = D^i$. Nous avons

$$C_1 = C; D^0 \setminus C^0; D^1 \setminus C^1; \cdots; D^{m-1} \setminus C^{m-1}$$

Nous vonlons montrer que C_1 et $\sigma^{(p)}(C_1)$ sont causalement équivalents, $p = 0, 1, \ldots, m-1$.

Par hypothèse, C est causalement équivalent à C^p. Par une permutation de C, C_1 est causalement équivalent à

$$C^p; D^0 \setminus C^0; \cdots; D^p \setminus C^p; \ldots; D^{m-1} \setminus C^{m-1}$$

En permutant $D^0 \setminus C^0$ et $D^p \setminus C^p$, nous obtenons

$$C^p; D^p \setminus C^p; \cdots; D^0 \setminus C^0; \cdots; D^{m-1} \setminus C^{m-1}$$

soit en fait

$$D^p; \cdots; D^0 \setminus C^0; \cdots; D^{m-1} \setminus C^{m-1}$$

C'est donc un calcul de la forme $D^p; D'$, où $P_{\sigma^{(p)}(a)}$ et $P_{\sigma^{(p)}(b)}$ sont inactifs dans D'. En appliquant σ, nous obtenons

$$D^{p+1}; D''$$

où $P_{\sigma(p+1)(a)}$ et $P_{\sigma^{(p+1)}(b)}$ sont inactifs dans D''. Donc la projection de $\sigma(C_1)$ sur ces deux processus est la même que celle de C_1, à savoir celle de D^{p+1}.

Ceci est vrai pour tout p. D'autre part, les processus inactifs dans $C_1 \setminus C$ le sont aussi dans $\sigma(C_1) \setminus \sigma(C)$ puisque C est équivalent à $\sigma(C)$ par hypothèse. Donc $\sigma(C_1)$ a mêmes projections que C_1 sur tous les processus. Par le théorème 2.3.4.1, $\sigma(C_1)$ est donc causalement équivalent à C_1. ∎

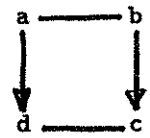


Figure 7.3.3.1.

Le théorème 7.3.3.1 s'applique par exemple à tout réseau invariant par permutation cyclique totale de ses sommets ("daisy-like networks"). Anneaux et graphes complets tombent dans cette catégorie. Dans ces cas, ce théorème est donc une généralisation du théorème 7.3.2.1. Aucun de ces réseaux n'admet de système électoral symétrique en CSP_{in}. Ce résultat mérite quelques commentaires. La raison profonde sous-jacente n'est pas seulement comme dans le cas du théorème 7.3.2.1, une symétrie excessive du système conduisant nécessairement à une possiblité de blocage général. Le résultat présent est d'une autre nature. Considérons le réseau bipartite présenté en figure 7.3.3.1.

Soit σ la symétrie par rapport à l'axe vertical. A chaque instant, la situation peut évoluer symétriquement dans P_a et P_b d'une part, et dans P_d et P_c d'autre part. P_a ne peut donc prendre aucune décision ultime et terminer, puisque, au même instant, P_b peut prendre la décision symétrique et lui aussi terminer dans un état inconsistant.

Il faut donc que P_a réussisse à briser sa symétrie avec P_b. Il ne peut le faire qu'en échangeant au moins un message avec P_b. Mais il ne peut le faire directement. En effet, si P_a se met en attente d'émission vers P_b, P_b pourrait, au même instant, se mettre lui aussi en attente d'émission vers P_a, et ceci produirait un blocage irrémédiable (lemme 7.3.3.3)! P_a doit donc passer par l'entremise de P_d. Mais, au même instant, P_b peut décider lui aussi de passer par l'entremise de P_c avec des intentions strictement symétriques (lemme 7.3.3.4)!

Dans ce cas, l'excès de symétrie conduit à la divergence par soucis d'éviter le blocage. Aucun processus ne peut prendre de décision ultime en toute sécurité, puisqu'il ne peut contrôler directement les intentions de ses partenaires. Ce type de comportement ne semble pas avoir été remarqué dans les travaux cités plus haut.

7.4. Le problème de l'élection en $CSP_{i/o}$

Nous avons étudié le problème de l'élection symétrique en $CSP_{i/o}$ et CSP_{in}. Nous avons vu que la symétrie, en l'absence de gardes d'émission, introduit des blocages indésirables. De plus, les stratégies permettant alors d'éviter ces blocages conduisent généralement à des calculs infinis. Cet état de fait s'explique par le caractère *engageant* (ou insistant) des communications. Un processus qui souhaite émettre ne peut le faire qu'en s'engageant irrémédiablement sur cette action. Cependant, il ne dispose pour prendre cette décision que d'informations partielles sur l'état des autres processus, puisque l'information ne peut être acquise précisément que par les communications. Pour éviter toute possibilité de blocage, les processus s'abstiendront donc en général d'émettre s'il en ont la possibilité. Mais, ce faisant, ils se privent de tout critère de décision, ce qui conduit aux calculs infinis cités plus haut. Il en va tout autrement dans le cas de $CSP_{i/o}$ où les gardes d'émissions sont admises. Une utilisation judicieuse des gardes d'émission et de réception permet alors de briser la symétrie par des moyens purement symétriques. Un degré excessif de symétrie ne conduit donc plus fatalement au blocage comme dans le cas précédent. Emettre n'est plus une action exigeant un engagement irrémédiable. Nous montrerons qu'une classe étendue de graphes admet des systèmes d'élection symétriques en $CSP_{i/o}$. Cependant, contrairement peut-être à ce qu'une lecture rapide laisserait penser, cet engagement n'élimine pas toutes les causes de divergence.De fait il reste possible que les processus se montrent incapables de réunir assez d'informations précises sur l'état du système pour prendre une décision définitive. Le problème réside alors, non plus dans la trop grande symétrie du système qui provoque des décisions incompatibles, mais dans le manque de moyens de synchronisation ou de communication entre processus.

```
P_u ::  [ sent:= false; received:= false; winning:= true; color:= white;
          *[ ▌ in ¬received[in]; P_in?pebble(col)
             -> if (winning ∧ (col = white))
                then winning:= false; winner:= in; color:= black end
                received[in]:= true
             ▌ out ¬sent[out]; P_out!pebble(color)
             -> sent[out]:= true
          ]]
```

Figure 7.4.1.1.

7.4.1. Résultat positif

L'exemple ci-dessous illustre comment un agencement judicieux des gardes d'émission et de réception permet de rompre la symétrie par des moyens purement symétriques. Soit G le graphe complet sur deux sommets. Par le théorème 7.3.2.1, G n'admet aucun système d'élection symétrique en CSP_{in}. Cependant, on vérifie aisément que le système suivant est un système d'élection symétrique pour G en $CSP_{i/o}$.

```
P = [P_a ‖ P_b]
P_a ::  [ P_b!pebble() -> decision:= a
        ▌ P_b?pebble() -> decision:= b]
P_b ::  [ P_a!pebble() -> decision:= b
        ▌ P_a?pebble() -> decision:= a]
```

Par la sémantique de CSP présentée au chapitre 1, ce système admet exactement six calculs maximaux (en fait, terminés). Dans chacun d'eux, exactement un des processus émet alors que l'autre reçoit, ce qui détermine un vainqueur (l'émetteur) et un vaincu (le récepteur). Nous allons montrer que cet exemple peut se généraliser à une famille très large de graphes. Nous présentons ci-dessous les notions de forêt recouvrante complète et de rigidité. Soit G un graphe orienté et F une forêt recouvrant G. F est **complète** s'il n'existe pas de côté dans G liant deux racines d'arbres de F. On remarquera que deux forêts recouvrantes complètes peuvent être strictement ordonnées pour l'inclusion. Un graphe G est rigide si pour toute forêt recouvrante complète F il existe un sommet u invariant par tout automorphisme de G laissant F invariant.

Théorème 7.4.1.1. *Soit $\mathcal{G}$ la famille des graphes fortement connexes rigides. Il existe une solution modulaire et homogène au problème de l'élection symétrique pour $\mathcal{G}$ en $CSP_{i/o}$.*

Le principe de l'élection est le suivant. Chaque processus P_u est prêt à envoyer exactement un jeton blanc ou noir à chacun de ses voisins sortants ou à recevoir exactement un jeton blanc ou noir de chacun de ses voisins entrants. Au début, P_u n'accepte d'envoyer que des jetons blancs, mais il accepte de recevoir des jetons blancs ou noirs. P_u est dit *vainqueur* s'il envoie un jeton blanc ou reçoit un jeton noir. Tant que P_u est vainqueur, il continue ainsi. P_u est *vaincu* par la réception d'un jeton blanc. Il enregistre alors le nom de son vainqueur. Par la suite, les jetons qu'il envoie sont de couleur noire. P_u termine lorsqu'il a échangé exactement un jeton sur chaque canal adjacent. L'algorithme est présenté en figure 7.4.1.1.

Lemme 7.4.1.1. *Tous les calculs de P terminent proprement. A la terminaison, P_u a échangé exactement un jeton sur chaque canal adjacent.*

Fixons un calcul maximal C de P. Soit F le graphe défini de la manière suivante. Les sommets de F sont ceux de G. F a un côté $u \longrightarrow v$ si, dans l'état final de C, $winning_v$ est faux et $winner_v$ est égal à u. Comme P_v ne peut perdre que par réception d'un message, F est un sous-graphe de G.

Lemme 7.4.1.2. *F est un forêt recouvrante complète de G.*

Preuve On remarque que la variable *winning* est monotone, et que *color* en est directement fonction. Montrons que F n'a pas de cycle. Supposons que

$$u_0 \longrightarrow u_1 \longrightarrow \cdots \longrightarrow u_p \longrightarrow u_0$$

soit un cycle de F. Soit λ_i la transition de P_{u_i} par laquelle il a été vaincu alors qu'il était vainqueur jusque-là (les indices sont pris modulo p). Nécessairement λ_i est postérieure à λ_{i+1}, car P_{u_i} a été vainqueur de $P_{u_{i+i}}$ avant d'être lui-même vaincu par $P_{u_{i-1}}$. Par transitivité, on obtient une contradiction.

La monotonie de *winning* implique qu'un processeur a au plus un voisin entrant dans F, et F est donc une forêt (recouvrante). Montrons qu'elle est complète. Soit u et v deux racines d'arbres de F liées par un canal au moins dans G. Soit $u \longrightarrow v$ l'un de ces canaux. Puisque C est maximal, exactement un message *pebble*(*color*) est émis par P_u sur ce canal dans C. Par monotonie de *winning*, P_u et P_v avaient leur variable *winning* mise à *true* à ce moment. Donc *color* valait *white* et donc nécessairement la réception de ce message par P_v a provoqué le passage de $winning_v$ à *false*. Donc P_v ne peut être une racine de F. ∎

Lemme 7.4.1.3. *P est symétrique. Un calcul de P contient au plus $N-1$ communications (N est le nombre de processeurs).*

Preuve La symétrie de P est obtenue par les conditions suffisantes développées en section 4.3.3, avec l'action évidente des automorphismes sur l'environnement. Remarquons que pour les applications S_σ exhibées ci-dessus, nous avons la propriété suivante. Si un calcul maximal C conduit à la construction d'une forêt F, S_σ conduit à $\sigma(f)$.

Lemme 7.4.1.4. *Toute forêt recouvrante F de G peut être obtenue par un calcul de P.*

Preuve Il suffit d'ordonnancer les communications en commençant par les feuilles de F. ∎

Comment procéder à l'élection sur la base d'une forêt recouvrante complète F ainsi produite? Supposons que nous disposions d'une application *decide* faisant correspondre un sommet à chaque forêt recouvrante complète. Supposons que *decide* vérifie pour tout automorphisme σ de G

$$decide(\sigma(F)) = \sigma(decide(F))$$

Nous pouvons convenir par exemple que le processus élu sera précisément celui de nom $decide(F)$. Cette décision respecte bien la symétrie.

Lemme 7.4.1.5. *Un graphe G est rigide si et seulement si il existe une telle application decide.*

Preuve Supposons que *decide* existe. Il suffit alors de prendre $u = decide(F)$ pour satisfaire la rigidité de G.

Réciproquement, considérons l'ensemble $\mathcal{F}$ des forêts recouvrantes complètes de G. Considérons les orbites de cet ensemble sous l'action du groupe Σ_G des automorphismes de G. Pour chaque classe, choisissons un représentant F. Choisissons l'un des sommet u_F dont l'existence est garantie par la rigidité de G appliquée à F. Nous avons

$$\sigma(F) = F \Longrightarrow \sigma(u_F) = u_F$$

Posons $decide(F) = u_F$. Etendons cette définition à l'orbite de F en posant $decide(\sigma(F)) = \sigma(u_F)$. Il est facile de vérifier la consistance de cette définition. Par construction, pour tout σ et tout F,

$$decide(\sigma(F)) = \sigma(decide(F))$$

∎

Il existe donc une certaine fonction *decide* faisant correspondre à toute forêt recouvrante complète de tout graphe G rigide dont les sommets sont pris dans l'ensemble (dénombrable) $Name$ l'un de ses sommets. Nous sommes maintenant prêts à décrire une solution modulaire en $CSP_{i/o}$ au problème de l'élection symétrique pour la famille $\mathcal{G}$ des graphes rigides fortement connexes. Cette solution est obtenue par composition de trois phases modulaires et (fonctionnellement) symétriques.

Phase 1

Les processus exécutent le système décrit en figure 7.4.1.1. A la terminaison, chaque processus P_u construit la donnée suivante : si $winner_u = v$, le couple (v, u); sinon, par convention, ("I_am_the_best", u).

Phase 2

Les processus executent un algorithme modulaire et symétrique multi-diffusion en $CSP_{i/o}$ des données ci-dessus pour les graphes fortement connexes (section 6.5). A la terminaison, chaque processus construit, à partir des valeurs reçues, la *même* forêt recouvrante complète F.

Phase 3

Chaque processus exécute

< $decision := decide(F)$ >

Par les résultats généraux du chapitre 5, nous obtenons le résultat souhaité. L'homogénéité est obtenu par le théorème 7.2.3. ∎

La condition de rigidité est cependant difficile à vérifier dans la pratique, car elle suppose l'examen de toutes les forêts recouvrantes complètes de G. C'est pourquoi nous décrivons ci-dessous une condition suffisante beaucoup plus simple à vérifier.

Propriété 7.4.1.1. *Soit G un graphe satisfaisant la condition suivante. Pour tout sommet u et tout automorphisme σ,*

1) soit $\sigma(u) = u$;

2) soit il existe p tel que $\sigma^{(p)}(u) \neq u$ et G a un côté entre u et $\sigma^{(p)}(u)$. Alors G est rigide.

Preuve Soit F une forêt recouvrante complète de G. F contient au moins un arbre T. Soit u la racine de T et soit σ un automorphisme tel que $\sigma(F) = F$. Supposons que $\sigma(u) \neq u$. Alors, par hypothèse, il existe p tel que $\sigma^{(p)}(u) \neq u$. $\sigma^{(p)}(T)$ est un arbre de F de racine $\sigma^{(p)}(u)$. Il est donc distinct de T. Puisque F est complète, il ne peut y avoir du côté dans G entre les racines de T et $\sigma^{(p)}(T)$, ce qui contredit l'hypothèse. ∎

Ce thèorème montre que les graphes complets admettent un système d'élection symétrique en $CSP_{i/o}$. Il est facile de voir que les anneaux *unidirectionnels* dont la taille est un nombre *premier* satisfont la propriété ci-dessus. Ils admettent donc aussi un système d'élection symétrique en $CSP_{i/o}$. Remarquons que ces graphes n'admettent aucun système d'élection symétrique en CSP_{in}. Par contre, les anneaux *bidirectionnels* de longueur première ne vérifient pas la propriété ci-dessus. Une preuve directe montre cependant qu'ils sont rigides et admettent donc un système d'élection symétrique en $CSP_{i/o}$. Remarquons enfin que les conditions du théorèmes 7.3.1.1 impliquent trivialement la rigidité. Ceci est conforme à l'intuition puisque $CSP_{i/o}$ étend trivialement CSP_{in}. Cependant, elles n'impliquent pas la condition suffisante ci-dessus.

7.4.2. Résultat négatif

De même que le théorème d'existence pour $CSP_{i/o}$ étend celui donné pour CSP_{in}, l'on pourrait chercher à resteindre les théorèmes de non existence de CSP_{in}. Le cas du théorème 7.3.2.1 est sans espoir puisqu'il repose sur l'existence d'un blocage dû à l'absence de garde d'émission. De plus, ce théorème s'applique principalement aux graphes complets et aux anneaux pour lesquels un système d'élection symétrique en $CSP_{i/o}$ existe. Le cas du théorème 7.3.3.1 est beaucoup plus prometteur. En fait, le seul rôle de l'absence de garde d'émission était alors de garantir la disjonction des arbites O_a et O_b (Lemme 7.3.3.3). Il suffit donc d'ajouter une condition suffisante pour assurer ce point.

Théorème 7.4.2.1. *Soit G un graphe admettant un automorphime équilibré nontrivial σ. Supposons que, pour tout sommet u et tout entier p, G n'ait pas de côté entre u et $\sigma^{(p)}(u)$. Alors G n'admet aucun système d'élection symétrique en $CSP_{i/o}$.*

Preuve La preuve est la même que celle du théorème 7.3.3.1. Il suffit simplement de vérifier le lemme 7.3.3.3. Mais si O_a et O_b ne sont pas disjointes alors $b = \sigma^{(p)}(a)$ pour un certain p. Or puisque P_a émet vers P_b, G a un côté de a vers b. ∎

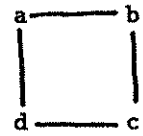


Figure 7.4.2.1.

Ce thèorème s'applique à tout anneau dont la longueur est un nombre *non premier*. Aucun anneau de cette forme n'admet de système d'élection symétrique en $CSP_{i/o}$. Les raisons en sont fondamentalement les mêmes que dans le cas de CSP_{in}. Considérons par exemple le cas du graphe présenté en figure 7.4.2.1.

A tout instant, P_c peut se comporter symétriquement par rapport à P_a. P_a ne peut alors prendre de décision et terminer puisqu'au même instant P_c peut prendre la décision symétrique et terminer. Le seul moyen serait pour P_a de briser sa symétrie avec P_c en lui envoyant un message. Mais P_a ne peut communiquer directement avec P_c. Il ne peut le faire que via P_b ou P_c. Supposons donc que P_a tente d'envoyer un message à P_c via P_b. Rien n'empêche P_c d'envoyer au même instant un message à P_a via P_b ! La divergence naît là encore du manque de possibilité de communication directe, comparée à la symétrie du réseau.

7.5. Applications

les résultats d'existence et de non-existence présentés ci-dessus sont certes intéressants en tant que tels. Ils montrent que, dans le cadres des systèmes répartis, il existe d'étroites relations entre des considérations topologiques d'une part, et sémantiques d'autre part. En effet, la topologie du graphe de communication détermine directement les possibilités de transfert d'information (de connaissance) au sein du système. Elle détermine donc la capacité des processus à coopérer malgré une connaissance partielle de leur état respectif. Si les transferts d'informations ne peuvent compenser l'incertitude des processus quant à l'état "réel" du système, ceux-ci ne peuvent coopérer correctement. Ils ne peuvent, en particulier, prendre les décisions irrémédiables mais nécessaires qui leur permettrait de progresser vers la solution.

7.5.1. Pouvoir expressif des gardes en *CSP*

Les résultats des section 7.3 et 7.4 nous donnent une évaluation du pouvoir expressif des gardes de communication en CSP. Plus précisément, définissons la notion d'implantion d'un dialecte A de CSP dans un autre dialecte B. C'est une application *Impl* qui à tout système réparti P du dialecte A fait correspondre un système réparti $impl(P)$ du dialecte B. Une implantation sera dite **répartie** si elle préserve le graphe de communication : P et $impl(P)$ sont bâtis sur le même réseau. Cette condition exprime que l'implantation n'utilise pas de processus annexe, par exemple pour centraliser toutes les demandes de communication et les apparier, comme c'est le cas dans l'implantation de CSP proposée par Richier [Richier 83]. D'autre part, cette condition garantit que le parallélisme du système initial n'est pas simulé, même de manière cachée, par une intercalation non déterministe des différentes étapes du calculs réalisée de manière centralisée. En fait, il est possible de simuler tout système réparti donné par un programme centralisé, séquentiel et non déterministe.

Ces conditions ne sont pas suffisantes, bien sûr. Une implantation doit respecter la sémantique des programmes (au moins dans une certaine mesure !). En fait, même au sein d'une implantation répartie, l'un des processus peut jouer de manière cachée le rôle de coordinateur décrit ci-dessus. Une condition supplémentaire est donc nécessaire pour spécifier qu'aucun processus n'est privilégié de cette manière. Nous dirons qu'une implantation répartie est **symétrique** si elle préserve la symétrie fonctionnelle : si P est fonctionnellement symétrique par rapport à une certaine action E, alors $impl(P)$ l'est aussi. Enfin, il nous faut exprimer qu'une implantation préserve la sémantique des programmes. Il est bien sûr trop fort d'imposer à une implantation de respecter les calculs. Il faut plutôt considérer le respect de certaines propriétés des calculs telles que

- la finitude,
- la terminaison,

- l'état final (*cf.* [AO 84] par exemple). Dans notre contexte il semble tout à fait indiqué d'exiger que l'implantation d'un système électoral soit un système électoral. Une implantation répartie sera donc dite **raisonnable** si elle préserve les systèmes électoraux. On remarquera la faiblesse de cette condition qui ne dit par exemple rien du traitement des calculs infinis, ou bien de la préservation des propriétés d'équité par l'implantation. Cependant, comme le montrent les théorèmes suivants, cette notion nous fournit un critère performant pour évaluer le pouvoir expressif des gardes de communications en CSP.

Théorème 7.5.1.1. *Il n'existe pas d'implantation symétrique raisonable de* $CSP_{i/o}$ *en* CSP_{in}.

Preuve Considérer le graphe complet sur 2 sommets. Il admet un système d'élection symétrique en $CSP_{i/o}$ et aucun en CSP_{in}. ∎

Remarquons qu'il existe de très nombreuses implantations raisonables de $CSP_{i/o}$ en CSP_{in}. De plus, ces implantations satisfont des propriétés de correction souvent très fortes comparées à celles considérées ici. Ces implantations ne peuvent être symétriques par le théorème ci-dessus; elles utilisent donc crucialement la notion de *priorité*. De fait, la plupart des protocoles de communication classiques tombent dans cette catégorie ([BS 83] par exemple). Considérons le système électoral symétrique déjà exhibé

$P = [P_a \parallel P_b]$

P_a :: [P_b?*pebble*() -> *decision*:= *b*
▮ P_b!*pebble*() -> *decision*:= *a*]

P_B :: [P_a?*pebble*() -> *decision*:= *a*
▮ P_a!*pebble*() -> *decision*:= *b*]

Une implantation répartie (non symétrique) en CSP_{in} satisfaisante de ce programme est par exemple

$P = [P_a \parallel P_b]$

P_a :: [[**true** -> *decision*:= *a*
▮ **true** -> *decision*:= *b*
]
P_b ! *pebble*(*decision*)]

P_b :: [P_a?*pebble*(*decision*)]

On voit que P_a impose son choix, ce qui se traduit par le fait que la seule communication possible est de P_a vers P_b.

Le théorème 7.5.1.1 exprime en fait une intuition présente dans nombreux travaux sur CSP ([Hoare 78], [Bernstein 80] *etc.*), à savoir que la présence de gardes d'émission augmente strictement la puissance de CSP. Cependant, il semble, à notre connaissance, que cette intuition n'avait, jusqu'à présent, pas reçue de preuve rigoureuse. Une question naturelle est alors de savoir si la notion de garde elle-même augmente strictement le pouvoir expressif de CSP. La réponse est aussi positive.

Théorème 7.5.1.2. *Il n'existe pas d'implantation symétrique raisonnable de* CSP_{in} *en* CSP_{no}.

Preuve Considérer le graphe bipartite complet à 2 et 3 sommets. Il admet un système électoral symétrique en CSP_{in} alors qu'il n'en admet pas en CSP_{no}. ∎

Les diverses formes de gardes de communication doivent donc être regardées comme primitives dans la sémantique de CSP en ce sens que la présence de chacune d'elle contribue à la puissance d'expression du langage. En fait, la notion de garde capture une certaine forme dégénérée d'équité. Considérons par exemple le système d'élection symétrique vu ci-dessus. Chacun des processus propose à l'autre d'être maître ou esclave. La sémantique de CSP montre qu'exactement l'une des deux possibilités sera finalement choisie. Ceci signifie que la négociation entre P_a et P_b se termine en un temps fini par un accord. Les deux partenaires

ayant des rôles rigoureusement symétriques, cette négociation ne eput être qu'une forme plus ou moins subtile de "pile ou face", chacun des processus lançant sa pièce jusqu'à ce que les valeurs obtenues soient différentes. La terminaison du programme ci-dessus correspond donc a la garantie que tout jeu de ce type termine en un temps fini, ce qui est très précisément une condition d'équité. Or, si la notion de garde capture une certaine forme d'équité, Hoare ([Hoare 78]) prend bien soin de préciser qu'aucune forme d'équité n'est supposée pour l'alternative non déterministe. Les résultats obtenus par Fauconnier ([Fauconnier 84]) montre que, en effet, il est impossible de simuler l'équité en CSP. Ces remarques expliquent maintenant clairement les résultats obtenus.

7.5.2. L'ω-équité ne peut être simulée en CSP

Les considérations ci-dessus nous amènent à étudier le type d'équité capturée par l'utilisation conjointe des gardes d'émission et de réception. Nous allons montrer que cette particularité sémantique de CSP ne permet d'obtenir qu'une forme dégénérée d'équité.

Pour chaque notion X d'équité, nous pouvons définir le langage CSP_X. Les programmes de CSP_X sont ceux de $CSP_{i/o}$ où les seuls calculs considérés sont précisément les calculs équitables pour la notion X. Nous considérons tout d'abord une notion extrêmement forte d'équité introduite par Eike Best ([Best 84]) à propos des réseaux de Petri. Soit $\langle P, env\rangle$ un état d'un système. Un état $\langle P', env'\rangle$ est dit **atteignable** à partir de cet état s'il existe un calcul (fini) ayant $\langle P, env\rangle$ pour état initial et $\langle P', env'\rangle$ pour état final.

Définition 7.5.2.1. *Soit*

$$C = \langle P_0, env_0\rangle \longrightarrow \langle P_1, env_1\rangle \longrightarrow \cdots$$

un calcul de P. C est ω-équitable si tout état atteignable à partir d'une infinité d'états $\langle P_i, env_i\rangle$ est effectivement atteint une infinité de fois par C. En particulier, tout calcul fini est ω-équitable. Nous montrons ci-dessous que cette notion garantit l'existence de systèmes d'élection symétriques.

Théorème 7.5.2.1. *Tout graphe G fortement connexe admet un système d'élection symétrique en CSP_ω.*

Pour exhiber un tel système, considérons le système électoral symétrique en $CSP_{i/o}$ décrit pour les graphes fortement connexes rigides en section 7.4.1. Appliquons ce système P à un graphe G fortement connexe quelconque. Tous ses calculs terminent proprement en construisant une certaine forêt recouvrante complète F. En fait toutes les forêts recouvrantes complètes de G sont atteignables.

Définition 7.5.2.2. *Une forêt recouvrante complète F de G est acceptable s'il existe un sommet u tel que $\sigma(F) = F$ implique $\sigma(u) = u$ pour tout automorphisme σ de G.*

Lemme 7.5.2.1. *Tout graphe fortement connexe admet une forêt recouvrante complète F qui est acceptable.*

Preuve Il suffit de prendre l'un des arbres recouvrants T. La racine de T est invariante par tout automorphisme laissant T globalement invariant. ∎

Comme en section 7.4.1, nous pouvons définir une application *decide* sur l'ensemble des forêts recouvrantes complètes acceptables de G. On peut alors considérer l'itération répartie du système P ci-dessus. A la fin de chaque phase, tous les processus ont acquis une copie de la même forêt recouvrante complète F. Ils testent alors si F est acceptable. Si c'est le cas, ils se décident sur $decide(F)$ et terminent. Sinon, il passent en phase suivante. Les théorèmes généraux du chapitre 5 montrent que le système Q ainsi obtenu est symétrique et modulaire pour la famille des réseaux fortement connexes.

Lemme 7.5.2.2. *Tous les calculs de Q en CSP_ω sont finis et terminent proprement.*

Preuve A chaque phase, il est possible d'obtenir chacune des forêts recouvrantes complètes acceptables de G. Par ω-équité, l'une d'elle sera obtenue dans tout calcul infini de Q. ∎

Il est alors facile de vérifier que Q est bien un système électoral symétrique pour G en CSP_ω. On en déduit alors le théorème suivant.

Théorème 7.5.2.2. *Il n'existe pas d'implantation symétrique raisonnable de CSP_ω en $CSP_{i/o}$.*

Preuve Considérer un anneau bidirectionnel composé de 4 processeurs. Il admet un système électoral symétrique en CSP_ω, mais aucun en $CSP_{i/o}$. ∎

Ce résultat s'explique en fait aisément si l'on se reporte à la discussion intuitive proposée en section 7.4.2. Les calculs où P_a et P_c agissent totalement symétriquement l'un par rapport à l'autre ne sont pas ω-équitables puisque les processeurs ont constamment la possibilité de cesser ce mimétisme. Sous l'hypothèse d'ω-équité, ces calculs ne peuvent perdurer et, tôt ou tard, P_a et P_c parviendront à briser leur symétrie. Le résultat reste en fait sous une hypothèse d'équité plus faible.

Définition 7.5.2.3. *Un calcul est K-équitable si tout état atteignable par un calcul de longeur au plus K est effectivement atteint une infinité de fois.*

On définit donc un dialecte CSP_K en ne considérant que les calculs K-équitables. De manière évidente, CSP_ω est un sous-langage de CSP_K, qui en est lui-même un de $CSP_{i/o}$ pour tout K. Il est clair que le nombre de transitions nécessaire pour compléter une phase du système Q ci-dessus est borné en fonction de la taille du réseau. On en déduit le corollaire suivant.

Corollaire 7.5.2.1. *Il n'existe pas d'implantation symétrique raisonnable de CSP_K en $CSP_{i/o}$ pour K assez grand.*

En examinant le cas de l'anneau à 4 sommets, $K = 100$ apparaît comme une valeur suffisante. Nous pensons cependant que le théorème devrait être vrai pour des valeurs de K beaucoup plus faibles. En fait, pour les systèmes dont le nombre d'états atteignables est fini, la K-équité est équivalente à la 1-équité*. Ce théorème montre que la forme d'équité introduite par l'usage des gardes d'émission et de réception reste relativement faible.

7.5.3. Equité des communications

Nous avons vu que la K-équité ne peut être implantée de manière symétrique et raisonnable en $CSP_{i/o}$. Ce résultat était attendu au vu de la discussion proposée en section 7.4.2. Nous allons voir que des comportements mimétiques sont cependant possibles en présence de formes plus faibles d'équité ([AFK 86]).

Définition 7.5.3.1. *Un calcul C est équitable pour les canaux si, pour tout couple de processus P_u et P_v, la condition suivante est satisfaite. Si P_u peut émettre vers P_v une infinité de fois dans C, alors il le fait.*

Cette équité sera appelée équité des communications. Le sous-dialecte de $CSP_{i/o}$ correspondant sera appelé CSP_{com}. CSP_{com} est une version affaiblie de CSP_K pour $K = 1$.

Théorème 7.5.3.1. *Soit G un graphe admettant un automorphisme équilibre non trivial σ. Supposons que pour tout sommet u et tout entier p, G n'a pas de côté entre u et $\sigma^p(u)$. Alors G n'admet aucun système électoral symétrique en CSP_{com}.*

Preuve Il suffit de raffiner la construction du théorème 7.3.3.1. Le seul problème est le choix des processus P_a et P_b. Remarquons que, dans le calcul infini C qui est construit comme limite des calculs C_i, les seules communications sont celles déduites de celle entre P_a et P_b. Le choix de ces processus étant arbitraire, il suffit de privilégier les canaux délaissés depuis le plus longtemps pour assurer l'équité. ∎

Notons que ce théorème n'implique pas qu'un graphe admette un système électoral symétrique en $CSP_{i/o}$ si et seulement si il en admet un en CSP_{com}. Dans le cas des graphes fortement connexes, ce serait vrai les théorèmes 7.4.1.1 et 7.4.2.1 étaient réciproques l'un de l'autre. Nous ne connaissons pas actuellement de telles conditions.

* Je dois cette remarque à Daniel Lehmann.

7.5.4. Application aux problèmes classiques

Nous avons vu que les résultats d'existence et de non existence de systèmes d'élection symétriques sont très fructueux sur le plan sémantique. Un autre intérêt de ces résultats, et sans doute non le moindre, est de fournir des solutions triviales à de nombreux problèmes classiques dans le domaine des systèmes répartis. Considérons par exemple les célèbres philosophes de Dijkstra ([Dijkstra 72]) qu'il est inutile sans doute de décrire ici. Dans la version originelle du problème, Dijkstra considère cinq philosophes assis autour d'une table. Or, par le théorème 7.4.1.1, un anneau bidirectionnel de longueur 5 admet un système électoral symétrique en $CSP_{i/o}$! Cette coincidence nous permet alors de décrire une solution symétrique suivante au problème.

1) Les philosophes exécutent une élection symétrique.
2) Le philosophe élu produit 4 jetons et les envoie successivement sur l'anneau, dans le même sens.
3) Un philosophe ne peut prendre ses fourchettes que lorsqu'il possède un jeton au moins.
4) Un philosophe est toujours prêt à recevoir un jeton.
5) Un philosophe ne peut manger deux fois consécutives sans avoir transmis au moins un jeton.

Il devrait être assez clair pour le lecteur que cette solution peut être écrite facilement en $CSP_{i/o}$, et que le système résultant est symétrique. Cette méthode s'applique en fait à toute tablée constituée d'un nombre *premier* de philosophes. On peut se demander pourquoi Dijkstra considère précisément 5 philosophes et non 6 ou 4. De fait, aucune justification n'est donnée de ce choix. On nous laissera supposer qu'en faisant ce choix, il pressentait que la primarité du nombre de philosophes peut être une aide pour gérer l'exclusion mutuelle de manière symétrique et équitable.

En fait, comme l'a montré le chapitre 5, ces résultats peuvent être utilisés pour rendre symétriques des solutions (dyssymétriques) déjà connues. Nous étudierons au chapitre 8 de manière détaillée le problème de la terminaison distribuée de Francez ([Francez 80]). Une solution pour un anneau de processeurs est décrites en [FRS 81]. Elle n'est pas symétrique car le processus P_0 y joue un rôle tout-à-fait particulier. C'est à lui que revient la charge d'initialiser les vagues de détection successives. Si l'anneau est de longueur première, il est possible de procéder tout d'abord à une élection symétrique pour déterminer le processus initiateur, et le système résultant sera symétrique. Il est bien évident que les solutions ainsi produites sont notoirement inefficaces, aussi bien en temps qu'en espace. Cependant, elles ont l'avantage d'être produites (et prouvées) de manière simple et directe par des méthodes générales. Pour de nombreux problèmes, l'existence même de telles solutions est resté longtemps indécise ! D'autre part, de telles solutions peuvent servir de base à des solutions optimisées en utilisant leurs contraintes spécifiques (topologie du réseau notamment). C'est volontairement que les algorithmes présentés ici ont été laissés sous une forme grossière, afin de mettre en valeur les méthodes qui ont permis de les produire.

8. Application à la détection de la terminaison répartie

Cette thèse est consacrée à l'étude des notions de modularité et de symétrie. Nous avons montré qu'une définition formelle de ces notions est possible, et qu'elle conduit à des résultats intéressants concernant le pouvoir expressif de certains langages de programmation répartie. Dans ce chapitre, nous souhaitons montrer que ces notions s'appliquent aussi à des problèmes concrets, soit pour évaluer les solutions existantes, soit pour servir de guide pour de nouvelles solutions. Nous considérons ici le problème de la détection de la terminaison répartie, proposé par Francez en 1980. Mis à part son intérêt pratique indéniable, ce problème présente l'avantage d'avoir été surtout étudié dans le cadre de *CSP*. Une douzaine de solutions différentes ont en effet été proposées dans ce langage et nous montrerons qu'aucune d'entre elles n'est satisfaisante selon nos critères. Nous décrirons donc une solution répartie modulaire symétrique et bornée pour ce problème. Le matériel de ce chapitre avait déjà été largement présenté dans [Bougé 85].

8.1. Le problème de la terminaison répartie

Le problème de la terminaison, répartie est intimement lié à celui de l'évolution de la connaissance au sein d'un système réparti. Il est donc important de présenter en détail cette notion.

8.1.1. Notion de connaissance

De nombreux travaux ([FHV 84], [HM 84], [Lehmann 84] *etc.*) ont attiré l'attention sur l'importance de la notion de connaissance dans le domaine des systèmes répartis. En effet, dans le cas séquentiel, le processeur peut, à chaque instant, accéder à toute l'information qui détermine les états sucessifs du système, à savoir l'environnement courant. D'un point de vue anthropomorphique, il maîtrise donc parfaitement l'évolution du système et en connaît tous les détails avec certitude. Il est "omniscient et omnipotent". La situation dans le cas réparti est tout-à-fait différente. Chaque processus est dans ce cas une machine séquentielle, et a donc pleine connaissance de son propre état. Par contre, un processus n'a qu'une connaissance partielle de l'état des autres processus. La connaissance est essentiellement *locale*. La connaissance non locale ne peut être modifiée que par le biais des interactions entre processus. La non séquentialité s'exprime ici par le fait que, du point de vue d'un processus, l'état des autres processus entre deux interactions est *incertain*.

Il n'est cependant pas totalement inconnu. En effet, les interactions entre processus obéissent à des règles strictes. Par exemple, dans le cadre des communications point-à-point, le processus qui reçoit un message sait qu'il a été émis à un certain point du calcul dans le passé. Dans les langages où le destinateur des messages est connu (*CSP* par exemple, mais non Ada), le processus sait aussi quel processus a émis ce message, ce qui restreint encore l'imprécision de sa connaissance quant à l'état courant de ce processus. Dans le cas de communications asynchrones, le gain d'information est donc relativement faible lors d'une communication, ce qui traduit précisément la notion de couplage faible entre processus. Par contre, dans le cas de communications synchrones, un processus qui reçoit un message sait que son destinateur est en train de l'émettre. Et réciproquement, le processus émetteur sait que le destinataire est en train de recevoir le message. Le couplage est donc ici beaucoup plus fort, et de plus, il introduit une certaine symétrie entre émetteur et récepteur. Il y a partage de connaissances plus que transmission. Cette remarque explique certaines particularités pathologiques de *CSP* décrites en section 1.2.

Nous n'avons jusqu'ici mentionné que la connaissance des faits objectifs, par exemple la position du contrôle ou la valeur de telle variable au sein de tel processeur. Mais il a été remarqué très tôt que la connaissance de degré supérieur, à savoir la connaissance sur la connaissance elle-même, joue aussi un très grand rôle. Il a par exemple été montré qu'une très large classe de problèmes, précisément ceux nécessitant la réalisation simultanée d'une action commune par des processus non fiables, se réduit en fait à l'acquisition de connaissances de degré suffisant ([DM 86] *etc.*) Par exemple, lors de l'échange synchrone d'un message entre deux processus, non seulement l'émetteur sait que le destinataire est en train de recevoir le message, mais il sait que le destinataire sait qu'il est en train de l'émettre. Un autre exemple est la terminaison des

systèmes répartis. Il est spécifié que le système ne termine que lorsque tous ses processus ont terminé. Pour qu'un processus puisse établir que le système est terminé, il faut donc qu'il sache que tous les processus ont terminé leur tâche, et de plus que tous les processus partagent cette même connaissance ! On parle alors de *connaissance commune* du système. Cette notion joue un rôle central dans les problèmes dits "byzantins".

Un autre aspect de cette approche est celui du *temps*. Dans un système réparti, les processus progressent à des vitesses arbitraires, les seules contraintes étant celles introduites par les interactions explicites, communications essentiellement. Une manière d'exprimer cette intuition est de considérer que le temps est lui aussi une connaissance locale ([Lamport 78]). Les processus ne peuvent alors acquérir qu'une connaissance relative du temps au sein des autres processus au moyen des synchronisations induites par les interactions : "avant", "après", et, dans le cas d'interactions synchrones, "en même temps". Cette remarque est souvent résumée en supposant l'absence d'horloge globale au système accessible par tous les processus. Cette situation a été comparée à celle issue de la mécanique quantique ([LeLann 77]). Il y a là un compromis entre le temps et l'espace qui rend toute connaissance imprécise et partielle.

Dans cette approche, la notion d'état global ou courant du système perd tout son sens puisqu'un tel état ne peut être perçu par les processus eux-mêmes. Il n'existe que pour une sorte d'observateur "divin" placé au-dessus du système (de tels observateurs divins sont appelés "daemons" en termes informatiques). Plusieurs observations différentes d'un même système sont compatibles avec les connaissances locales des processus mais non forcément identiques. Par exemple, si un processus peut faire l'action a et un autre l'action b indépendamment (concurremment), alors les observations $\langle a, b\rangle$ et $\langle b, a\rangle$ sont toutes deux consistantes. Les exemples suivants montrent l'importance de ces remarques en algorithmique répartie.

8.1.2. Terminaison répartie

Il existe de très nombreux cas partiques où les processus d'un système réparti doivent prendre conscience de l'état global du système qu'ils forment. Considérons par exemple une banque où travaillent de nombreux employés. Chaque employé gère un certain nombre de comptes. Au début de la journée les employés reçoivent l'ensemble des ordres exécuter concernant les comptes qu'ils gèrent. Ces ordres peuvent augmenter ou diminuer la somme déposée ou simplement transférer une partie de cette somme d'un compte à l'autre. Dans ce dernier cas, deux employés doivent en général coopérer pour envoyer et recevoir la somme.

Il se trouve que les employés travaillent dans différents bureaux et communiquent par téléphone. Supposons que l'employé A gère le compte X et que B gère Y. Pour effectuer un transfert vers B, A lui téléphone pour se mettre d'accord sur le transfert. A diminue alors X et B augmente Y de la somme considérée. Le règlement est qu'un employé peut rentrer chez lui dès son travail terminé. Cette règle est assortie d'un correctif. Les employés indisciplinés, absents alors que l'on a besoin d'eux et empêchant ainsi d'autres employés d'accomplir leur travail, sont immédiatement renvoyés, ce qu'aucun employé ne souhaite.

La conséquence est qu'aucun employé ne pourra jamais rentrer chez lui! En effet, considérons l'employé B ci-dessus. Supposons qu'il ait fini son travail. Il ne peut savoir si A l'a aussi terminé. S'il rentre à la maison, il se peut que, plus tard, A souhaite executer un ordre de transfert de X vers Y. A serait alors dans l'impossibilité de faire son travail à cause de l'absence de B! B doit donc rester et attendre, peut-être éternellement, un coup de téléphone de A! Bien sûr, il pourrait téléphoner à A pour lui demander ce qu'il en est. Mais, même si A lui répond que, pour le moment, il n'a pour le moment ordre de transfert en attente, il peut en découvrir plus tardldots Nous pouvons donc imaginer les employés assis à leur bureau, tard dans la nuit, attendant un coup de téléphone qui ne viendra jamais (*cf.* le film de Buzzati "Le désert des tartares"*).

Des situations similaires sont très courantes dans la pratique de l'algorithmique répartie. Reprenons l'exemple de la multi-diffusion présenté en section 0.1, et examinons la solution 4. Chaque processus diffuse à ses voisins toutes les nouvelles données qu'il posséde. Au bout d'un temps fini, tous les processus possèdent toutes les données. Cependant aucun d'eux ne sait qu'il possède toutes les données, et donc chacun reste prêt à recevoir de nouvelles données. Un blocage d'un type très particulier se produit. En effet, l'état final est, à strictement parler, atteint. Mais aucun processus ne peut prendre la décision irrévocable de terminer par manque de connaissance. Remarquons que chacun des processus est dans un état stable en ce sens qu'il

* Michel Raynal me signale qu'un problème analogue est traité par Brel dans la chanson "Je m'appelle Zangra et je suis Lieutenant". Daniel Lehmann me cite de plus "Le rivage des syrtes" de Gracq, et "En attendant Godot" de Beckett.

n'exécutera pas d'action sans avoir été auparavant "réveillé" par un autre processus. On peut dire qu'un état d'équilibre stable a été atteint par le système en ce sens qu'il se maintiendra tant qu'aucune interaction extérieure n'interviendra. Comment le problème est-il résolu dans la pratique ? Les employés disposent habituellement de montres qui peuvent être considérées comme parfaitement synchronisées. La convention est alors que tous les employés cessent leur travail à 17 heures. Ainsi personne n'est dérangé dans son travail par l'absence d'autrui. Cette solution nécessite l'accés à une horloge globale, mais aussi le fait que l'heure de sortie soit une connaissance commune : tout le monde sait que tout le monde sait que...que le travail se termine à 17 heures. Une telle solution n'est pas applicable aux systèmes répartis puisque, par hypothèse, le temps universel n'est pas accessible aux processus qui ne disposent que d'horloges locales. Nous serrons en section 8.2 qu'il est cependant possible de trouver des solutions. Elles sont appelées algorithmes de detection de la terminaison répartie par Francez ([Francez 80]).

8.1.3. Propriétés stables

En fait, le problème évoqué ci-dessus n'est qu'un cas particulier d'un problème plus général : la détection de propriétés de l'état global d'un système. Nous avons déjà vu en section 8.1.1. que les processus d'un système ne peuvent prendre conscience de leur état global avec une entière précision. Les exemples suivants illustrent ce point.

Supposons que le directeur de la banque veuille connaître la somme totale déposée sur les comptes. Pour cela, il téléphone successivement à chacun des employés, et lui demande la somme déposée sur ses comptes qu'il gère, puis il fait la somme de ces totaux partiels. Malheureusement, cette procédure naive peut produire des résultats inconsistants! Considérons les employés A et B ci-dessus, et supposons que le compte X se monte à 100, et Y à 200. Le directeur interroge A qui lui répond 100. Ensuite, pendant que le directeur interroge un autre employé C, B effectue un transfert de 100 de Y vers X. Lorsque le directeur interroge B celui-ci lui répond donc 100. Une somme de 100 a donc disparue "derrière de dos" du directeur!

Le problème est ici de reconstruire un état global à partir des états locaux. Un échantillonnage arbitaire de ces états locaux conduit en général à un résultat inconsistant. Il faut donc synchroniser correctement l'échantillonnage. Ceci a été appelé par Chandy et Lamport le problème du "snapshot" (cliché, instantané) ([CL 85]). La détection de la terminaison répartie en est bien sûr un cas particulier. Morgan ([Morgan 85]) a remarqué que ce problème est lié directement à l'absence d'horloge globale. Dans la pratique, en effet,il est facile d'instituer la règle que tous les jours à 10 heures les employés notent la somme totale déposée sur leurs comptes. Le directeur interroge ensuite chacun d'eux. Le total représente exactement la somme possédée par la banque à 10 heures. Remarquons que, même avec l'aide de cette procédure, le directeur n'obtient pas une vue instantanée de sa banque. Il ne peut détecter qu'un état global passé. Cependant, il est assuré que cet état a effectivement été celui de sa banque à 10 heures.

Si aucune horloge globale n'est disponible, il est cependant possible de simuler une telle synchronisation globale. Le directeur téléphone à chacun de ses employés et lui demande de cesser son travail. Il téléphone de nouveau à chacun pour lui demander son total. Il téléphone enfin une troisième fois à chacun pour lui permettre de reprendre son travail. L'ensemble des sommes partielles recueillies représente effectivement l'état de la banque à la fin de la première série de coups de téléphone. Cependant le directeur ne peut plus contrôler quel état est ainsi échantillonné. D'autre part, cet échantillonnage introduit une perturbation notable du système en éliminant certains de ces comportements. Du fait de l'asynchronisme des différents employés, il aurait pu se faire que l'état échantillonné ne soit jamais atteint par le système. Cette synchronisation parasite introduite dans le système est appelée *gel* ("freeze") par Francez. Les procédures fondées sur un tel gel sont dites *insistantes*, par opposition aux procédures *indulgentes* ([FR 82]). Nous reviendrons sur ce point en section 8.2.4.

Supposons que le directeur soit intéressé, non pas par l'état global en lui-même, mais seulement par une propriété de cet état. Nous avons vu qu'il ne peut contrôler, en l'absence d'horloge globale, le déroulement de l'échantillonnage. Si la propriété est verifiée seulement de manière intermittente il est possible qu'elle soit malheureusement invalide pour les échantillons successifs et valide par ailleurs. De telles propriétés ne peuvent être détectées avec certitude. Par contre, supposons que la propriété ne puisse être invalidée après avoir été valide. Alors, si la propriété est valide à un certain moment, tout échantillonnage initialisé strictement après cet instant la détectera! Ces propriétés sont appelées *stables* par Chandy et Lamport. De très nombreuses propriétés globales utiles des systèmes répartis sont stables. La terminaison répartie, et

plus généralement l'interblocage, sont des exemple de propriétés stables. Le but de l'algorithme du cliché est précisément le test de telles propriétés.

Les sections suivantes présentent un algorithme modulaire et symétrique pour détecter de telles propriétés dans les systèmes répartis avec communications synchrones. Cet algorithme s'applique donc en particulier à la détection de la terminaison répartie. L'idée fondamentale est que pour détecter une propriété stable, il suffit d'"imaginer" un état du système par lequel le système aurait pu passer pour parvenir à l'état courant. Un tel état est appelé cliché ("snapshot"). Si la propriété à tester est valide pour un cliché, elle est valide dans l'état courant. Il suffit donc de savoir faire coopérer les processus pour reconstruire un cliché global à partir d'informations locales. Pour reprendre les mots de [Dijkstra 83]

> *the purpose of the snapshot algorithm is to collect such (local) information that, on the account of it, (global) stability can be detected.*

8.2. L'algorithme des clichés répétés

Nous présentons ci-dessous l'algorithme original de Chandy et Lamport ([CL 85]) adapté au cas des communications synchrones, puis la généralisation que nous en proposons.

8.2.1. Historique et modèle

L'algorithme des clichés ("snapshot algorithm") est semble-t-il apparu pour la première fois dans [Chandy 83]. La description est très brève. Chandy considère des communications asynchrones et une sémantique partiellement ordonnée. Une descrition plus précise est donnée par [Dijkstra 83], dans le cadre de machines abstraites avec des communications asynchrones. Dijkstra utilise une sémantique entrelacée. Il étudie deux versions de l'algorithme. La première ajoute une "couleur" blanche ou rouge aux machines et aux messages pour synchroniser échantillonnages et interactions. La seconde implante ces couleurs par des signaux appelés *marqueurs*. Un exposé détaillé est aussi proposé par [Lamport 84], dans le cadre d'une sémantique partiellement ordonnée. Cet exposé inclut une discussion de la notion d'état global d'un système réparti qui prolonge celle de [Lamport 78] et [Dijkstra 83]. Enfin, l'article de référence est [CL 85]. Il considère des communications asynchrones et une sémantique entrelacée.

Morgan ([Morgan 85]) a montré que la notion de cliché est liée à celle d'horloge globale. En fait, une succession de clichés peut être vue comme la simulation d'une horloge globale à partir d'horloges locales. Les versions ci-dessus de l'algorithme peuvent donc être vue comme la composition de deux algorithmes distincts

1) une version modifiée de l'algorithmes du cliché où une horloge globale est accessible par tous les processus; sa correction est alors triviale;
2) un algorithme de synchronisation qui simule une sorte d'horloge globale à partir d'horloges locales ([Lamport 78]).

Aucun des travaux ci-dessus n'aborde le problème des clichés répétés d'un système. En fait, le contrôle additionnel nécessaire pour éviter les interférences entre les clichés successifs dépend crucialement du modèle considéré. Notre travail, qui se place dans le cas de communications synchrones, est original de ce point de vue. Il semble que, dans le cas asynchrone, il n'y ait pas, à notre connaissance, de moyen élégant de procéder (le problème est seulement mentionné dans [Lamport 84] par exemple).

Dans la suite de cette section, nous considérons un modéle analogue à celui des systèmes répartis du chapitre 2. Cependant, l'algorithme est indépendant de la structure interne des processus. On pourra donc les considérer comme des "machines" ([Dijkstra 83]) abstraites, capables de recevoir ou d'envoyer des messages (systèmes d'automates communicants). A chaque pas du système, une machine modifie sa mémoire privée ou deux machines échangent un message le long d'un canal du réseau. Nous utiliserons dans cette section les notions présentées au chapitre 2: restriction et extension de calculs, projection, fusion, commutation de pas concurrents *etc.*

Nous repérons un état global global S par un calcul fini C ayant cet état comme état final. Rappelons que par calcul nous entendons toujours calcul partiel non nécessairement maximal. Soit B un prédicat sur l'état global d'un système. $B(C)$ signifie que B est valide pour l'état final de C. Remarquons que si C et C' sont causalement équivalents, alors $B(C)$ et $B(C')$ sont équivalents. La stabilité peut alors être définie ainsi.

Définition 8.2.1.1. *Une propriété B est dite stable si, pour tous calculs finis C_0 et C_1 tels que C_0 soit une restriction de C_1, $B(C_0)$ implique $B(C_1)$.*

En fait, tous les résultats de cette section s'applique à des prédicats sur les calculs finis stables par équivalence causale et par extension. A notre connaissance, cette remarque n'a cependant pas encore trouvé d'application.

8.2.2. L'algorithme de Chandy et Lamport

Examinons, dans l'exemple de la section 8.1.3., la raison pour laquelle le directeur obtient une image inconsistante de sa banque. Les transferts de fonds peuvent être de trois types :

1) Les deux employés n'ont pas encore échantillonné leur état au moment du transfert ;
2) Les deux l'ont fait ;
3) L'un des deux seulement l'a fait.

Seuls les tranferts du troisième type sont susceptibles de provoquer des inconsistances. Le problème est donc d'interdire de tels transferts, ou, plus généralement, de synchroniser échantillonnages et communications. A chaque transfert, les employés doivent donc vérifier qu'ils ne se trouvent pas dans le cas 3. Sinon, l'employé qui n'a pas encore échantillonné son état doit le faire avant d'effectuer le transfert. Ce sera cet état qu'il transmettra au directeur lorsqu'il sera, éventuellement bien plus tard, interrogé à son tour.

Cette idée peut être directement appliquée aux systèmes répartis que nous considérons . Soit P un tel système. Nous ajoutons aux processus de P la possibilité d'exécuter un certain nombre d'actions de contrôle en plus des actions primitives dictées par l'algorithme original. Le système est alors dit **équipé**, par opposition au système original qui est dit **nu**. Notre règle de conduite est que le contrôle peut espionner et retarder (éventuellement indéfiniment) les actions du processus primitif mais non les modifier ou interférer avec elles. Le processus primitif ne peut "prendre conscience" de la présence du contrôle. On dit alors que le contrôle est *surimposé* au processus primitif, en ce sens qu'il filtre ses actions sans les perturber.

Nous surimposons donc à chaque processus le contrôle nécessaire pour échantillonner son état local primitif. Un processus est dit *blanc* s'il n'a pas encore échantillonné son état local primitif, et *rouge* sinon. Le contrôle doit garantir qu'un message primitif ne peut être échangé qu'entre deux processus de même couleur. Comment implanter cette restriction? L'idée de Chandy et Lamport est de donner au contrôle la possibilité d'échanger un tye spécial de message, appelé *marqueur*, le long des canaux du réseau. Les marqueurs sont utilisés pour forcer le partenaire à échantillonner son état, et ainsi permettre à une communication primitive d'avoir lieu de manière consistante. Les régles sont alors les suivantes.

Règles du marqueur

1) Initialement, tous les processus sont blancs. Un processus devient rouge en échantillonnant son état local primitif.
2) Un processus blanc peut devenir rouge spontanément. Un processus rouge ne peut devenir blanc.
3) Un processus blanc est prêt à recevoir des marqueurs. Sur réception d'un marqueur, il devient rouge.
4) Un processus rouge est prêt aussi à recevoir des marqueurs, mais il les ignore. Un processus rouge envoie un nombre fini de marqueurs;
5) Un processus rouge n'échange un message primitif sur un canal que lorsque qu'il a échangé au moins un marqueur sur ce canal.

Soit C un calcul (partiel) du système P équipé. Le **support** de C est la suite de transitions obtenue en oubliant tout ce qui concerne le contrôle. Le lemme suivant montre que la discipline décrite ci-dessus satisfait nos critères de surimposition.

Lemme 8.2.2.1. *Le support d'un calcul du système équipé est un calcul du système nu.*

Preuve Les actions primitives des processus ne peuvent être que retardées par la règle 5, et le contrôle n'accède qu'en lecture à l'état primitif. ∎

Lemme 8.2.2.2. *Les messages primitifs ne sont échangés qu'entre processus de la même couleur.*

Preuve Considérons une communication primitive du processus P_a vers P_b. Si P_b est rouge, il ne peut recevoir un message avant qu'un marqueur ait été reçu de P_a, et donc avant que P_a soit lui-même devenu rouge. Réciproquement, si P_a est rouge, par la règle 5 un marqueur a déjà été envoyé vers P_b. Puisque les communications sont synchrones,il a donc déjà été reçu et P_b est rouge par la règle 3. ∎

Les propriétés suivantes montrent que les règles ci-dessus permettent effectivement de prendre des clichés du système. Notons S_u l'état local primitif échantillonné par P_u lorsqu'il devient rouge (noter qu'il ne le devient qu'une fois). Soit SSS (SnapShot State) l'état global obtenu en juxtaposant les états locaux S_u.

Propriété 8.2.2.1. *Soit C un calcul du système P équipé où tous les processus deviennent rouges, et soit SSS l'état ainsi échantillonné. Alors il existe un calcul fini C_0 du système nu, d'état final SSS, support d'une restriction d'un calcul C' causalement équivalent à C.*

En d'autres termes, l'état global SSS est un état par lequel le système aurait pu passer. Mais, de toute façon, il est impossible pour les processus de savoir s'il y est effectivement passé ou non. C'est en ce sens qu'un cliché peut être vu comme un état global passé consistant *imaginé* collectivement par les processus d'un système.

Preuve ([Dijkstra 83])

Preuve Remarquons d'abord que, par le lemme 8.2.2.2, les processus actifs dans une transition, sauf peut-être un échange de marqueur, sont tous de la même couleur. On peut donc parler de la couleur d'une transition, avec la convention qu'une transition correspondant à un échange de marqueur est rouge.

Il existe une restriction finie C_1 de C telle que tous les processus soient rouges dans l'état final de C_1. Supposons qu'au sein de C_1 une transition blanche suive une transition rouge. Nécessairement, par la règle 2, ces transitions sont concurrentes. Par un nombre fini de commutations, on obtient un calcul C_2 où les transitions blanches précèdent les transitions rouges. Soit C_3 la restriction de C_2 à ses transitions blanches. Soit C_0 le support de C_3. L'état échantillonné est précisément l'état final de C_0. ∎

La propriété suivante exprime que la surimposition du contrôle ne restreint pas le comportement du système primitif.

Propriété 8.2.2.2. *(complètude) Soit C_0 un calcul fini du système P nu, d'état final S. Alors il existe un calcul C du système équipé, de suppport C_0, produisant précisément le cliché S.*

Preuve Le système équipé se comporte comme suit. Le contrôle est d'abord inactif, et les processus exécutent les actions primitives de C_0. Lorsque C_0 est terminé, chaque processus envoie un marqueur à chacun de ses voisins. Chaque processus échantillonne alors son état primitif, ce qui produit précisément le cliché S. ∎

Il reste à vérifier que les actions accomplies par le contrôle ne submergent pas l'algorithme primitif. C'est donc une propriété d'équité de leurs progrès respectifs.

Propriété 8.2.2.3. *(maximalité) Le support d'un calcul maximal fini du système équipé est un calcul maximal fini du système nu. Le support d'un calcul infini du système équipé est un calcul infini du système nu.*

Preuve Soit C un calcul fini maximal du système équipé. Puisque les processus peuvent devenir rouges spontanément, ils sont tous rouges dans l'état final de C, et au moins un marqueur a été échangé sur chaque canal. Soit C_0 le support de C. Si C_0 n'est pas maximal, l'un des processus peut exécuter une action primitive dans l'état final de C_0. Cette même action est encore possible dans l'état final de C: contradiction.

Si C est infini on remarque qu'un nombre fini de marqueurs seulement peut être échangé par la règle 4. C_0 est donc infini, et donc maximal. ∎

Remarquons qu'il reste possible que le contrôle ne progresse pas au cours d'une exécution infinie. Une condition d'équité additionnelle, spécifique à l'application considérée, serait nécessaire. La propriété suivante montre que l'algorithme ci-dessus permet de détecter les propriétés stables des systèmes répartis.

Propriété 8.2.2.4. *(détection) Soit B une propriété stable des états d'un système P. Soit C un calcul fini du système P équipé où tous les processus deviennent rouges, soit C_0 son support et soit SSS le cliché résultant. Si B est valide pour l'état SSS, alors B est valide dans l'état final de C_0.*

Preuve Par la propriété 8.2.2.1, il existe un calcul C_1 de P d'état final SSS qui est une restriction d'un calcul C_2 causalement équivalent à C_0. Puisque B est stable, $B(C_1)$ implique $B(C_2)$ et donc $B(C_0)$. ∎

Remarquons que la réciproque n'est pas vraie. B peut être valide pour C_0 sans l'être pour SSS. Ceci se produit par exemle si le cliché est pris "trop tôt" dans le calcul C. Il est facile cependant de vérifier que si B est valide dans un état où tous les processus sont encore blancs, alors B sera detectée par le cliché SSS.

8.2.3. Clichés répétés

L'algorithme de Chandy et Lamport décrit ci-dessus n'est cependant pas entièrement satisfaisant. Si la propriété B ne devient valide qu'au cours de l'échantillonnage, aucune garantie n'est fournie sur sa détection. Revenons à notre exemple de la section 8.1.2. Les employés peuvent essayer d'utiliser l'algorithme ci-dessus pour détecter s'ils ont tous terminé leur travail, ce qui est une propriété stable. Si, par malheur, des employés trop pressés de rentrer chez eux envoient trop tôt leurs marqueurs, le résultat pourra être négatif alors qu'en fait le travail est réellement terminé. Il faut donc que les employés puissent répéter cette procédure. L'une des itérées sera initialisée dans un état où tous ont terminé, et la stabilité sera alors détectée par la propriété 8.2.2.4.

Il nous faut donc modifier les règles du marqueur énoncées en section 8.2.1. Le problème est de blanchir les machines. Mais il faut prendre garde aux interférences entre les vagues successives de marqueurs. Un marqueur émis par une machine qui a déjà été k fois rouge doit être reçu par une machine dans un état similaire. Il s'agit donc en fait de construire l'itération répartie de l'algorithme de Chandy et Lamport au sens du chapitre 5, en garantissant la synchonisation des différentes phases. La méthode proposée au chapitre 5 revient à fixer le nombre de messages (ici, messages de contrôle) échangés sur chacun des canaux dans chacune des phases. Cette méthode a l'avantage de ne pas utiliser d'estampillage non borné des messages. La remarque fondamentale est que les processus rouges n'ont en fait besoin d'envoyer qu'un seul marqueur sur chaque canal, les suivants étant ignorés. Nous pouvons donc modifier la règle 2 de la manière suivante.

Règle du marqueur (suite)

2') Un processus blanc peut devenir rouge spontanément. Un processus rouge envoie au plus un marqueur sur chaque canal sortant, et en accepte au plus un sur chaque canal entrant. Un processus rouge qui a échangé exactement un marqueur sur chaque canal devient blanc un jour.

Nous décrivons ci-dessous les propriétés de l'algorithme ainsi modifié. Comme au chapitre 5, nous définissons la notion de phase, dépendant cette fois-ci seulement du contrôle et non de l'algorithme primitif.

Définition 8.2.3.1. *Un processus est dans sa phase p s'il est déjà devenu rouge p fois.*

Lemme 8.2.3.1. *Si P_u quitte sa phase k, il a échangé exactement k marqueurs sur chacun de ses canaux adjacents.*

Preuve P_u échange exactement un marqueur sur chaque canal à chaque phase. ∎

Lemme 8.2.3.2. *(synchronisation) Si P_u et P_v échangent un message primitif, alors ils sont dans la même phase.*

Preuve Si P_u envoie un message primitif à P_v sur un canal alors qu'il est dans sa phase k, il a déjà envoyé à P_v exactement k marqueurs sur ce canal. Les communications étant synchrones, P_v les a reçus, et est donc au moins en phase k. S'il était en phase $k+l$, il ne pourrait accepter de message de P_u avant d'avoir reçu un nouveau marqueur par la règle 5. Il est donc en phase k. ∎

Ce lemme garantit la synchronisation des phases successives, et donc la consistance de l'itération répartie. Soit SSS_k l'état global obtenu en combinant les états locaux échantillonnés au début de la phase k. Les propriétés ci-dessous généralisent celles vues en section 8.2.2.

Propriété 8.2.3.1. *(correction) Soit C un calcul du système équipé où tous les processus entrent en phase p au moins, et soit SSS_k, $k = 1, ldots, p$, les états ainsi échantillonnés. Il existe un calcul C', causalement équivalent à C, et des calculs C_k, $k = 1, ldots, p$, du système nu, tels que*

$$C_1 \prec C_2 \prec \cdots \prec C_p \prec support(C')$$

et SSS_k est l'état final de C_k, $k = 1,\ldots,p$.

Preuve On procède comme pour la propriété 8.2.2.1. On se restreint d'abord à un calcul C fini. Par le lemme 8.2.3.2, on peut associer une phase à chaque transition. Deux transitions successives telles que la seconde soit d'une pphase inférieure à celle de la première sont alors nécessairement concurrentes. Par un nombre fini de commutations, on obtient un calcul C' où les phases des transitions sont croissantes. C_k est alors le support de la restriction de C' à ses transitions de phase strictement inférieure à k. SSS_k est exactement l'état final de C_k. ∎

Propriété 8.2.3.2. *(complètude) Soit*

$$C_1 \prec C_2 \prec \cdots$$

une suite (éventuellement infinie) strictement croissante de calculs du système nu, restrictions d'un calcul C_0, d'états finaux S_1, S_2, ... Il existe un calcul C du système équipé, de support C_0, produisant successivement les clichés $Ssub1$, S_2, ...

Preuve Le système équipé se comporte comme suit. Le contrôle reste d'abord inactif, et les processus exécutent les actions (primitives) de C_1. A la fin de C_1, tous deviennent rouges spontanément, échangent un marqueur sur chaque canal, puis redeviennent blancs. Ils poursuivent alors le calcul C_2 *etc.* ∎

La formulation des propriétés de détection de la stabilité est délicate, puisque rien ne garantit l'équité entre les transitions du contrôle et celles de l'algorithme primitif.

Propriété 8.2.3.3. *Soit B une propriété stable du système P, et C un calcul du système équipé où une infinité de clichés sont pris. Soit C_0 le support de C. B est valide pour un état de C_0 si et seulement elle est détectée dans l'un au moins des clichés de C.*

Preuve Soit C'_0 un sous-calcul fini de C_0 tel que B soit valide dans l'état final de C'_0, et soit C' un sous-calcul fini de C, de support C'_0. Dans C', chaque processeur est actif dans au moins une transition à chaque phase. Il existe donc au moins une phase initialisée strictement après la fin de C' dans C. Le cliché produit dans cette phase est un successeur de l'état final de C_0, et donc B y est valide. La réciproque découle directement de la propriété 8.2.3.1. ∎

Propriété 8.2.3.4. *(maximalité) Le support d'un calcul maximal fini (resp. infini) du système équipé est un calcul maximal fini (resp. infini) du système nu.*

Preuve La preuve est très semblable à celle de la proppriété 8.2.2.3. Le point crucial est que dans un calcul maximal fini, les processus sont tous blancs dans l'état final. ∎

Cette propriété ne garantit cependant pas la progression du contrôle. Un cas important est celui où la condition stable B est en fait *bloquante* (c'est en particulier la cas des propriétés "quiescent" selon le vocabulaire de [CM 85]). La terminaison répartie a évidemment cette caractéristique.

Définition 8.2.3.2. *Une propriété B est bloquante si pour tous calculs C_0 et C_1, $B(C_0)$ et $C_0 \prec C_1$ implique que C_1 est fini.*

En d'autres termes, si B est valide à un point du calcul, alors ce calcul du système nu finira par se bloquer.

Tout le problème est donc de garantir qu'une infinité de clichés peuvent être pris, tout en garantissant que la maximalité des calculs primitifs est préservée. Il s'agit donc de forcer une sorte d'équité entre le contrôle et l'algorithme primitif. Une possibilité est de ne permettre au contrôle d'initialiser une nouvelle vague qu'à certains points du calcul primitif, de telle manière que celui-ci ne puisse être submergé par les flots de marqueurs. Nous renforçons donc la règle 2' de la manière suivante.

Règles du marqueur (suite)

2″) Un processeur peut devenir rouge spontanément seulement s'il a effectué une transition primitive depuis la dernière fois où il est devenu rouge spontanément (si cela s'est jamais produit).

Sous cette règle, il y a au plus autant de vagues de marqueurs que de transitions primitives. Remarquer que la propriété 8.2.3.2 reste valide puisque la suite est supposée strictement croissante. Remarquons que, si B est valide, non seulement le calcul du système nu, mais aussi celui du système équipé finit par se bloquer. Ce blocage se produira au plus N phases après celui du support. Au moment du blocage, tous les processus seront blancs, en attente de réception d'un marqueur. Le dernier cliché pris sera exactement l'état final du système nu, et donc le blocage de celui-ci pourra être détecté par le contrôle. Cette remarque joue un rôle fondamental dans l'application décrite en section 8.4. Nous pouvons résumer cette discussion par la propriété suivante.

Propriété 8.2.3.5. *Soit B une propriété stable et bloquante d'un système P. B est valide si et seulement si elle est détectée.*

8.2.4. Evaluation et discussion

L'algorithme que nous avons présenté permet de detecter les propriétés stables et bloquantes. Exactement E signaux (marqueurs) sont échangés au cours de chaque phase, et il y a au plus une phase par transition de l'algorithme primitif. Aucun processeur n'est privilégié dans la détection de la propriété B, ce qui se traduira par la *symétrie* de l'implantation en CSP présentée en section 8.3. Aucune information concernant la topologie globale du réseau n'est nécessaire, ce qui se traduira par la *modularité* de l'implantation. Remarquer, que, strictement parlant, le réseau n'a pas besoin d'être connexe. La connexité ne sera utilisée que pour la diffusion des échantillonnages locaux.

Un point important dans l'évaluation de ce type d'algorithme est le degré d'entrelacement entre le contrôle et l'algorithme primitif. Dans notre cas, cet entrelacement n'est pas arbitraire. Nous avons déjà remarqué ce phénomène en section 8.1.3. Il est appelé *gel* ("freezing") par Francez ([FR 82]).

Définition 8.2.4.1. *(indulgence) Soit P un système équipé d'un mécanisme de contrôle. Le contrôle est dit indulgent si pour tout calcul C du système équipé la propriété suivante est satisfaite. Soit C_0 le support de C. Pour toute extension C'_0 de C_0, il existe une extension C' de C , de support C'_0, telle que $C' \setminus C$ ne contienne aucune communication de contrôle.*

En d'autres termes le contrôle peut à tout instant rester passif sans perturber l'algorithme primitif. La progression du contrôle est facultative. En ce sens, notre algorithme de détection n'est pas indulgent. Il est dit *insistant*. En effet, si un processus est rouge, il ne peut recevoir un message primitif sur un canal sans avoir auparavant reçu un marqueur sur ce canal. Il existe certaines situations où l'algorithme primitif ne peut progresser que si le contrôle a auparavant progressé. Cependant, les propriétés 8.2.3.2 et 8.2.3.4 montrent que le contrôle ne peut introduire de perturbations ni de blocages dans l'algorithme primitif.

Cette insistance peut être vue comme un défaut de notre algorithme. Ce problème semble inhérent à la notion de cliché. A notre connaissance il n'existe pas d'algorithme indulgent de detection de propriétés stables générales. Pour certains cas restreints, notamment le cas de la terminaison répartie avec indicateurs locaux, il existe cependant des algorithmes indulgents (*cf.* [SF 86] pour un exemple très élégant).

8.3. Implantation en CSP

Notre but est ici de décrire une implantation de l'algorithme précédent dans le cadre du langage CSP.

8.3.1. Spécification

Dans ce cadre formel rigoureux, nous pouvons donner une spécification précise de cet algorithme. Nous cherchons en fait une transformation

$$P \mapsto T(P)$$

qui fait correspondre à tout système réparti P en CSP (le système nu) un autre système $T(P)$ (le système équipé avec le contrôle additionnel). Indépendamment de l'algorithme implanté, plusieurs exigences peuvent être définies pour une telle transformation.

Exigences (préservation)

1) $T(P)$ est construit sur le même réseau que P (répartition).
2) Il existe une application
$$C \mapsto support(C)$$
qui fait correspondre à tout calcul C de $T(P)$ un calcul $support(C)$ de P et qui préserve l'équivalence causale et la restriction des calculs (correction).
3) Cette application est surjective: pour tout calcul C_0 de P il existe un calcul C de $T(P)$ tel que C_0 soit le support de C (complétude).

Les trois exigences ci-dessus expriment que $T(P)$ est obtenu en ajoutant du contrôle à P sans le modifier. $T(P)$ est une extension de P qui "préserve" P, pour reprendre le vocabulaire des types abstraits algébriques. Nous pouvons ensuite caractériser la qualité de cette extension.

Exigences (qualité)

4) Si P est symétrique alors $T(P)$ l'est aussi (symétrie).
5) Soit $\mathcal{P}$ une famille modulaire. Soit $T(\mathcal{P})$ le famille des systèmes $T(P)$, pour P dans $\mathcal{P}$. Alors $T(\mathcal{P})$ est modulaire (modularité).
6) Si toutes les variables de P sont bornées, alors celles de $T(P)$ aussi (extension bornée).

Il nous faut ensuite exprimer que le contrôle ne peut introduire de blocage dans l'algorithme original.

Exigences (vivacité)

7) Le support d'un calcul fini est fini.
8) Le support d'un calcul infini est infini.
9) Le support d'un cacul maximal est maximal.
10) Le support d'un calcul terminé est terminé.

Nous exprimons maintenant la notion de détection d'une propriété stable dans le cadre de CSP. Du fait de l'atomicité très fine de CSP, tous les états ne sont pas également significatifs. Par exemple dans la séquence

x := 1 ; y := 2

nous pouvons considérer que l'état du processus entre les deux affectations n'est pas significatif, ce qui revient à considérer cette séquence comme atomique.

< x := 1 ; y := 2 >

Seuls les états liés aux synchronisations seront significatifs.

Ceci nous conduit donc à supposer que tous les processus sont dans la forme normale présentée au chapitre 3,

P_u :: [*Init*
*[$Guard_1$ -> $Command_1$
▮ ...
▮ $Guard_p$ -> $Command_p$
]]

où toutes les commandes de communication apparaissent dans les gardes $Guard_i$, $i = 1,\ldots,p$. Nous dirons qu'un état d'un processus en forme normale est significatif si le contrôle est situé devant sa boucle principale ou après celle-ci (si la processus est en fait terminé). Un état global d'un système est **significatif** s'il est formé d'états locaux significatifs. Le lemme suivant montre l'importance de cette notion.

Lemme 8.3.1.1. *Soit C un calcul fini d'un système P en forme normale où tous les processus P_u de P exécutent au moins leur partie $Init_u$. Alors C est causalement équivalent à une concaténation $C_0; C_1$ telle que l'état final de C_0 est significatif et C_0 est maximal.*

Le calcul C_0 peut être vu comme la partie significative de C. Remarquons que C_0 n'est pas nécessairement un sous-calcul de C. Dans notre implantation, la commutation entre transitions de l'algorithme primitif et du contrôle ne se produit qu'aux états significatifs. Ceci est revient donc à considérer que l'exécution d'une commande gardée est *atomique*.

Nous supposons que chaque processus du système $T(P)$ déclare une nouvelle variable $detected_u$. Nous dirons qu'une propriété est **détectée** si $detected_u$ a la valeur *true* dans l'état courant pour tous les processus.

Exigences (détection)

11) Si B est detectée dans l'état final de C, alors B est valide pour l'état final de $support(C)$.
12) Soit C_0 un calcul fini de P d'état final vérifiant B. Soit C un calcul de $T(P)$ de support C_0. Alors C admet une extension C' telle que B soit détectée dans l'état final de C'.

Il faut remarquer que la propriété 12 ne garantit pas que B sera détectée dans tous les cas. Elle garantit que ce sera le cas sous une hypothèse d'ω-justice: tout état continûment atteignable est atteint une infinité de fois. En particulier tout calcul fini est évidemment ω-juste.

Comment garantir les exigences 1 et 2? Une méthode simple est de travailler sur la forme syntaxique des processus en ne permettant qu'un certain nombre de modifications. Considérons un processus P_u d'un système P en forme normale.

P_u :: [*Init*
*[$Guard_1$ -> $Command_1$
▮ ...
▮ $Guard_p$ -> $Command_p$
]]

Nous nous restreignons donc à construire des système $P' = T(P)$ où le processus P_u ait la forme normale suivante

P_u :: [$Init'$; *Init*
*[$Bool_1$; $Guard_1$ -> Cmd_1; $Command_1$
▮ ...
▮ $Bool_p$; $Guard_p$ -> Cmd_p; $Command_p$
▮ $Guard'_1$ -> $Command'_1$
▮ ...
▮ $Guard'_q$ -> $Command'_q$
]]

En d'autres termes, nous nous autorisons à préfixer les parties séquentielles, à préfixer les gardes par des booléens et à rajouter d'autres commandes gardées. De plus, toutes les variables accédées en écriture (c'est-à-dire dans les ensembles WV de la section 1.3) dans ces parties ajoutées doivent être nouvelles. Les variables de contrôle sont donc des variables auxiliaires par rapport à celles de l'algorithme primitif. De même, les types des messages apparaissant dans les nouvelles gardes $Guard'_j$, $j = 1, \ldots, q$ doivent être nouveaux. Nous dirons que P' est obtenu par **surimposition** des constructions de contrôle au système primitif P.

Sous ces restrictions, nous pouvons définir le support C_0 d'un calcul C de $T(P)$ par induction de manière analogue à la définition de la projection en section 2.3.1. Intuitivement, C_0 est obtenu à partir de C en effaçant toutes les transitions concernant de nouvelles gardes $Guard'_j$, de nouvelles commandes $Command'_j$, $Init'$ ou Cmd_i.

Lemme 8.3.1.2. *C_0 est un calcul de P.*

Preuve On remarque que les transitions effacées de degré 1 ne peuvent modifier l'environnement de P. Les transitions effacées de degré 2 ne peuvent concerner que de nouvelles gardes puisque les types des messages ajoutés sont nouveaux. ▮

Remarquons que ceci ne garantit absolument pas les autres exigences ci-dessus. En particulier, la procédure ci-dessus peut conduire à l'effacement d'un nombre infini de transitions. D'autre part, rien n'interdit de prendre $Bool_i = false$, ce qui contredit en général l'exigence 3.

8.3.2. Implantation des marqueurs

La règle 1 est implantée par une nouvelle variable booléenne *red*. Elle vaut initialement $false$. Elle vaut *true* si le processus est rouge. L'échantillonnage de son état par un processus se fait par l'appel de la fonction $SAMPLE$. L'état local est conservé dans une nouvelle variable *state*. La définition exacte de $SAMPLE$ et de *state* est évidemment dépendante du problème spécifique considéré. Nous en verrons un exemple en section 8.4. La possibilité de devenir spontanément rouge (règle 2) est exprimé par la nouvelle commande gardée suivante.

(8.3.2.1)

$\neg red$ -> *state*:= $SAMPLE$; *red*:= **true**

Nous implantons les marqueurs par des signaux d'un nouveau type, *marker*. Réceptions et émissions sont consignées dans deux tableaux *received*[*In*] et *sent*[*Out*]. D'après la règle 2', au plus une communication se fait sur chaque canal à chaque phase. Nous pouvons donc considérer des tableaux de booléens, initialisés à $false$. La règle 3 est donc implantée par la commande gardée suivante.

$\mathbin{\mathbf{I}}_{in}$ $\neg red$; $P_{in}?marker(\ )$
-> *state* := $SAMPLE$; *red* := **true** ; *received*[*in*] := **true**

La règle 2' est implantée par les commandes gardées suivantes.

$\mathbin{\mathbf{I}}_{in}$ *red* ; $\neg$*received*[*in*] ; $P_{in}?marker(\ )$
-> *received*[*in*]:= **true**
$\mathbin{\mathbf{I}}_{out}$ *red* ; $\neg$*sent*[*out*] ; $P_{out}!marker(\ )$
-> *sent*[*out*]:= **true**

La règle 5 exprime l'influence du contrôle sur le déroulement de l'algorithme primitif. L'exécution de certaines transitions peut être empêchée par le contrôle. Ceci est implanté en préfixant les gardes correspondantes par une expression booléenne. Remarquer que ceci est possible parce que ces transitions correspondent à des communications. Une telle implantation ne serait pas possible dans le cas de transitions qui ne peuvent être exprimées par une garde (transitions non significatives). Chaque commande gardée par une réception de l'algorithme primitif

Bool ; $P_{in}?message$ -> *Command*

est préfixée de la manière suivante

(8.3.2.2)

(*received*[*in*] $\vee \neg red$) ; *Bool* ; $P_{in}!message$ -> *Command*

De même, chaque commande gardée par une émission

Bool ; $P_{out}!message$ -> *Command*

devient

(8.3.2.3)

(*sent*[*out*] $\vee \neg red$) ; *Bool* ; $P_{out}!message$ -> *Command*

Enfin, les commandes gardées seulement par des booléens restent inchangées.

Pour implanter la règle 2″, il nous faut garder trace des transitions de l'algorithme primitif. Du fait de la forme syntaxique que nous considérons, l'occurrence d'une telle transition est nécessairement associée à l'éxecution de *Init*, au passage de l'une des gardes $Guard_i$ ou à l'exécution de l'une des commandes $Command_i$.

Puisque dans notre implantation un processus ne peut devenir rouge que lorsque le contrôle se trouve devant sa boucle externe, il suffit d'enregistrer les exécutions des commandes gardées de l'algorithme primitif, et non de toutes les transitions. Il suffit donc d'ajouter une nouvelle variable booléenne *moved*. Elle vaut *true* si au moins une transition primitive a eu lieu depuis la dernière fois que la commande gardée 8.3.2.1 a été exécutée si elle l'a jamais été. La commande gardée 8.3.2.1 devient alors

```
▌ ¬red ; moved
-> state:= SAMPLE ; red:= true ; moved:= false
```

et chaque commande $Command_i$ est prefixée pour devenir

```
moved:= true ; Command_i
```

Enfin, il nous faut permettre aux machines de redevenir blanches selon la règle 2′. Ceci est réalisée par la commande gardée suivante.

(8.3.2.4)

$$\blacksquare\ red\ ;\ \bigwedge_{in} received[in]\ ;\ \bigwedge_{out} sent[out]$$
$$\rightarrow\ red := \texttt{false}\ ;\ received := \texttt{false}\ ;\ sent := \texttt{false}$$

8.3.3. Diffusion des échantillonnages

La version abstraite de l'algorithme des clichés répétés n'a pas traité le problème de la diffusion des échantillonnages locaux dans le réseau. Il faut permettre en effet à tous les processeurs d'apprendre le même cliché et ainsi faire des choix cohérents. Nous pouvons utiliser pour faire cette diffusion l'algorithme de multi-diffusion vu en section 6.5. Ici, nous devons diffuser une suite de données (les échantillons successifs) produites dynamiquement. En remarquant que cet algorithme de multi-diffusion n'a aucun calcul équivoque au sens de la section 5.1.3, on peut appliquer les résultats généraux et considérer son itération répartie. Remarquer que tous les processus acquièrent le même ensemble de données à chaque phase, précisément le cliché obtenu à cette phase. Selon les méthodes vues en section 6.2.3, ceci permet de forcer la terminaison de l'itération répartie en garantissant la synchronisation des expressions booléennes commandant l'itération. Il nous suffit maintenant de surimposer cet algorithme à celui obtenu dans la section 8.3.2. La synchronisation est garantie en combinant la condition de blanchissement 8.3.2.4 et la condition d'arrêt de la multi-diffusion. Il faut aussi garantir que l'apprentissage dynamique de la donnée produite localement par chaque processus (son échantillonnage local) est correctement pris en compte. En remarquant que cette valeur n'est apprise qu'une seule fois, il suffit de remplacer

```
state:= SAMPLE
```

par

(8.3.3.1)

$$state := SAMPLE\ ;\ set := set \cup \{(u, state)\}\ ;\ forwarded := \texttt{false}$$

La forme finale de la transformation ainsi obtenue est décrite en figure 8.3.1. La fonction $DETECTED(set)$ teste si la condition B est valide pour le cliché *set* du système. L'exigence 9 découle de la propriété 8.2.3.4. Dans l'état final d'un calcul maximal fini, le contrôle dans chacun des processus est devant la boucle externe, *moved* et *red* valent *false* dans chacun d'eux, et les tableaux *received*, *sent*, *forwarded*, *completed* sont aussi tels que toutes leurs cellules contiennent la valeur *false*. La dernière valeur prise par *set* avant d'être

```
                                                    /* the transformed algorithm */
P_u : : [
                                                /* initialization of control variables */
    received:= false; sent:=false; completed:= false; forwarded:= false;
    moved:= true; halt:= false; snapshot:= ∅;
                                                        /* the original Init part */
    Init
                                                            /* the main while loop */
    *[
                                                /* the original boolean guarded commands */
    ▌ ¬halt; bool -> moved:= true; command
                                                /* the original input guarded commands */
    ▌ ¬halt; (received[in] ∨ ¬red); bool; P_in?message -> moved:= true; command
                                                /* the original output guarded commands */
    ▌ ¬halt; (sent[out] ∨ ¬red); bool; P_out!message -> moved:= true; command
```

Figure 8.3.1. *(suite ci-dessous).*

réinitialisé est précisément l'état final du système primitif dans le support du calcul. Cette remarque sera cruciale pour les résultats de la section 8.4.

L'exigence 10 par contre n'est pas satisfaite par la forme présente de l'implantation. Elle sera traitée en section 8.4. Pour l'instant, les calculs terminés du système produisent des calculs de $T(P)$ qui se bloquent dans l'état décrit plus haut. L'exigence 11 découle de la propriété 8.2.3.3. L'exigence 12 est la plus importante. Supposons que la propriété B soit valide dans l'état final du support C_0 d'un calcul fini maximal C. Deux cas sont possibles. Soit B est détectée dans l'état final de C, et l'exigence est trivialement satisfaite. Soit B n'est pas détectée et donc aucun cliché n'a été initialisé strictement après la dernière transition de C_0. Mais le processus actif dans cette transition est alors dans un état où sa variable *moved* a la valeur *true*. Il peut donc initialiser un nouveau cliché qui détectera B. Remarquer qu'il n'est pas vrai que B sera nécessairement détectée dans un calcul infini. Si dans un calcul infini, le contrôle dans l'un des processus reste dans une des commandes primitives $Command_i$, aucun cliché ne pourra être pris. Cependant, puisque, par hypothèse, les commandes $Command_i$ ne peuvent ni diverger ni provoquer d'erreur, la propriété 8.2.3.5 reste valide.

8.4. Détection de la terminaison répartie

Nous montrons dans cette section que l'implantation décrite en section 8.3 conduit directement à une solution pour le problème de la détection de la terminaison répartie. Le problème posé par Francez ([Francez 80]) est le suivant. Soit G un réseau fortement connexe et P un système réparti sur G, supposé en *forme normale*. On suppose que tous les calculs de P sont finis et sans erreur. On suppose de plus que pour chaque processus P_u, on connaît une expression booléenne B_u construite sur les variables de P_u qui satisfasse la propriété suivante.

```
                                                        /* to turn red spontaneously */
▌ ¬halt; moved; ¬red
-> state:= SAMPLE; UPDATE({(u, state)}); red:= true; moved:= false
                                                    /* to turn red on receiving a marker */
▌ ¬halt; ¬red; P_in?marker()
 in
-> state:= SAMPLE; UPDATE({(u, state)}); red:= true; received[in]:= true
                                                  /* to receive a marker once red */
▌ ¬halt; ¬received[in]; red; P_in?marker()
 in
-> received[in]:= true
                                                        /* to send a marker once red */
▌ ¬halt; ¬sent[out]; red; P_out!marker()
 out
-> sent[out]:= true

                                                         /* to receive a snapshot */
▌ ¬halt; ¬completed[in]; P_in?info(new_snapshot)
 in
-> UPDATE(new_snapshot)
                                                       /* to send one's snapshot */
▌ ¬halt; ¬forwarded[out]; P_out!info(snapshot)
 out
-> forwarded[out]:= true

                                                       /* to whiten or terminate */
▌ ¬halt; red;
  ⋀ received[in]; ⋀ sent[out]; ⋀ completed[in]; ⋀ forwarded[out]
  in      out      in           out
-> [ DETECTED(snapshot) -> halt:= true
   ▌ ¬DETECTED(snapshot) -> received:= false; sent:=false;
                            completed:=false; forwarded:= false;
                            snapshot:= ∅; red:= false
   ]
]]
```

Figure 8.3.1. *(suite).*

Hypothèse

Soit C un calcul de P. C est maximal si et seulement si dans l'état final de C, pour tout processus P_u, la condition suivante est satisfaite. Le contrôle dans P_u se trouve devant sa boucle externe ou après celle-ci (dans le cas où P_u est terminé) et B_u s'évalue à *true*.

On pose

$$B = \bigwedge_u B_u$$

Il est clair que B est stable. B est de plus bloquante (en fait "quiescent" selon le vocabulaire de [CM 85]) de manière évidente puisque tous les calculs sont supposés finis. Une telle propriété est appelée "locally-indicative stable" dans [SF 86]. La propriété 8.2.3.5 s'applique donc ici, et l'implantation de la section 8.3 peut être utilisée. La fonction $SAMPLE$ produit simplement la valeur de l'expression B_u, et la fonction $DETECTED$ produit la conjonction de ces valeurs; L'état local d'un processus consiste donc en un seul bit dans ce cas.

Nous obtenons ainsi une solution modulaire pour les réseaux fortement connexes, symétrique et bornée pour le problème de la détection de la terminaison répartie. Soit M le nombre de transitions du calcul; il y a au plus $O(M)$ vagues de détection (clichés). Pour chacune d'elle, il y a $O(E)$ marqueurs échangés, plus $O(N * E)$ messages de diffusion des états échantillonnés, où N est le nombre de processeurs et E le nombre de canaux. On remarquera qu'en factorisant l'exploration du réseau dans la première vague, et en adoptant une convention convenable pour éviter d'indexer chaque état local par le nom du processus correspondant, on peut ramener la taille de ces messages de diffusion à une longueur N.

Cette solution a été, à notre connaissance, la première à satisfaire de telles propriétés ([Bougé 84]). Elle doit être comparée avec les nombreuses autres solutions proposées pour ce problème. La solution originelle de Francez ([Francez 80]) en CSP est fondée sur un arbre recouvrant le long duquel sont collectées et diffusées les informations. Cette solution n'est pas symétrique puisque la racine de l'arbre joue le rôle de moniteur. Elle n'est pas modulaire puisque tous les processus doivent connaître leur position dans l'arbre avant de pouvoir initialiser l'algorithme. Enfin, l'algorithme utilise un gel explicite de tous les processus pour garantir la consistance des informations collectées. Ce dernier point est amélioré dans ([FR 82]), où un algorithme *indulgent* est proposé. Une version de cet algorithme qui utilise un circuit hamiltonien au lieu d'un arbre recouvrant est présentée en [FRS 81]. Dijkstra et Scholten ([DS 80]) proposent une solution modulaire valable dans tout réseau connexe. Elle n'est pas symétrique, puisqu'une machine spéciale, modélisant l'environnement du système, initialise et détecte la terminaison. Cependant, le degré de dissymétrie est bien moindre que ci-dessus, puisque cette machine ne joue un rôle spécial qu'au début et la fin de la détection. Rana ([Rana 83]) semble avoir été le premier à proposer un algorithme symétrique. Sa solution a ensuite été *corrigée* et améliorée dans [AR 84] et [Richier 84] dans le cadre de CSP. Ces solutions présupposent la connaissance d'un circuit hamiltonés dans le réseau, et ne sont donc par modulaires. Ces solutions sont directement dérivées de l'algorithme général de Dijkstra, Feijen et van Gasteren ([DFG 83]).

Notre solution présente cependant plusieurs défauts. Tout d'abord, sa complexité est très élevée, puisqu'elle utilise $O(M * N * E)$ messages. D'autre part, elle est *insistante* selon le vocabulaire [FR 82], comme nous l'avons vu en section 8.2.4. Ces divers défauts ont été corrigé par Shavit et Francez ([SF 86]), en reprenant les idées de [DFG 83]. Leur algorithme est modulaire pour les réseaux fortement connexes, symétrique et borné. Sa complexité est en $O(M)$ et il est totalement indulgent. Cependant, cet algorithme dépend crucialement de la forme $B = \bigwedge_u B_u$ de la condition de stabilité, et d'une propriété additionnelle qui est que si B_u est vraie alors P_u ne peut émettre de messages. Le problème résolu est donc une spécialisation du problème que nous envisageons, et il ne semble pas possible d'étendre leur solution à un cadre plus général. Il nous semble donc que, malgré les défauts cités plus haut, notre approche garde l'avantage de la généralité des problèmes considérés.

9. Conclusion

Notre travail peut se résumer par la formule suivante.

Trouver des critères d'évaluation des algorithmes mesurant spécifiquement leur degré de répartition.

La modularité et la symétrie ne sont en fait que deux exemples de tels critères. Ils répondent sans doute très partiellement au problème posé, mais il nous semble que ce travail montre que de tels critères peuvent être définis, étudiés rigoureusement, et utilisés avec profit. La notion de répartition quitte ainsi le domaine du sentiment et de l'intuition pour entrer dans celui de l'objectivité et de la rigueur.

Comment poursuivre cette recherche? Il est clair que l'une des faiblesses de notre travail est son cadre. Nous avons longuement commenté le choix du langage CSP. La principale motivation était l'existence d'une définition rigoureuse du langage et d'un large corpus d'algorithmes "typiques" des problèmes que nous nous proposons d'étudier. Mais CSP est un langage déjà ancien, et certaines de ses caractéristiques apparaissent aujourd'hui discutables. Nous avons cependant tenu à ne pas modifier le langage à notre convenance, pour que nos résultats soient directement applicables. Ceci nous a conduit à certaines difficultés techniques. Une autre critique est que CSP propose un type très particulier de répartition et d'interaction qui est loin d'être représentatif.

Il est donc nécessaire de reprendre dans d'autres cadres les notions définies ici. Il nous semble que les principales questions sont les suivantes.

1) Comment les notions de modularité et de symétrie peuvent-elles être définies *raisonnablement* compte tenu des spécificités des modèles considérés? (chapitres 3 et 4)
2) Existe-t-il des méthodes de construction permettant de garantir la modularité et la symétrie des algorithmes produits? (chapitre 5)
3) Existe-t-il des méthodes permettant de modulariser et symétriser des algorithmes déjà existants? (chapitre 5)
4) Comment ces notions permettent-elles de mieux cerner le pouvoir expressif du modèle? (chapitres 6 et 7)
5) Comment appliquer concrètement ces notions à des algorithmes de taille réaliste? Comment ceci permet-il de meiux comprendre des problèmes déjà connus par ailleurs? (chapitre 8)

Chacune de ces questions est importante. Une recherche comme celle-ci n'a de sens que si elle peut être appliquée avec profit par les algorithmiciens (*cf.* section 3.3 et 4.4.3).

Quels modèles choisir? Bernadette Charron ([Charron 86]) a étudié le cas de CCS ([Milner 80]). Au moins dans le cas de la symétrie, des réponses satisfaisantes peuvent être apportées aux questions ci-dessus. En fait, il appert que la situation est très proche de celle de $CSP_{i/o}$, notamment en ce qui concerne l'existence d'algorithmes d'élection symétriques. Il faut donc s'orienter vers des modèles plus éloignés de CSP. Il nous semble que deux voies sont plus particulièrement prometteuses : les modèles avec communications asynchrones, et ceux avec création dynamique de processus.

Il est clair que les notions de modularité et de symétrie ne permettent pas d'exprimer parfaitement l'intuition de la répartition. Par exemples, les algorithmes de diffusion des sections 6.2 et 6.3 sont tous modulaires et symétriques. Cependant, le premier semble, au moins intuitivement, "vraiment" réparti, alors que le second n'est qu'un parcours séquentiel et centralisé du réseau simulé dans un univers réparti. Nous avons ici besoin de critères permettant d'évaluer des notions aussi subtiles que le degré de parallélisme, le degré de non déterminisme, la résistance aux perturbations externes (pannes, mais aussi ralentissement de certains processeurs *etc.*) par exemple. Ceci reste un domaine à peine exploré encore.

Enfin, notre étude laisse un problème fondamental non résolu. D'une part, ces notions de modularité et de symétrie semblent "universelles", en ce sens qu'elles sont liées à la notion-même de répartition, et non au modèle. D'autre part, nous ne savons les définir et les étudier que dans le cadre d'un modèle précis, dont les caractéristiques techniques ont une influence déterminante. Des notions universelles *devraient* pouvoir

être exprimées dans un langage universel! Nous devons convenir que nous ne voyons pour le moment pas de solution à ce problème. Un candidat naturel serait peut-être les récentes théories de la connaissance qui permettent de décrire l'évolution d'un système indépendamment des caractéristiques du modèle d'interaction utilisé. De la résolution de ce problème dépendent crucialement les suites qui pourront être données à cette étude. Il y certainement là un terrain prometteur pour une recherche de longue haleine.

10. Références

[**Angluin 80**] D. Angluin, Local and global properties in networks of processes, Proc. 12*th* Ann. ACM Symp. Theory Comp., Los Angeles, Calif. (1980) 82-93.

[**Apt 83**] K.R. Apt, A static analysis of CSP programs, Rept. No. 83-34, LITP, Univ. Paris 7, Paris (1983).

[**AC 85**] K.R. Apt, Ph. Clermont, Two normal form theorems for CSP programs, Research Report, IBM Research Center, Yorktown Heights, New York (1985).

[**AF 84**] K.R. Apt, N. Francez, Modelling the distributed termination convention of CSP, ACM Trans. Prog. Lang. Syst. 6, 3 (1984) 370-379.

[**AKF 86**] K. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Proc. ACM Symp. Princ. Progr. Lang., Münich (Janvier 1986).

[**AO 84**] K. Apt, E.-R. Olderog, Transformations realizing fairness assumptions in parallel programs, invited paper in: M. Fontet, K. Melhorn, eds., Proc. 1*st* Symp. on Theor. Aspects of Comp. Science, Lect. Notes in Comp. Science 166 (Springer, 1984) 26-42.

[**AR 84**] K. Apt, J.-L. Richier, Real time clocks versus virtual clocks, Rept. No. 84-34, LITP , Univ. Paris 7, Paris (1984), to appear in ACM Trans. Prog. Lang. and Syst.

[**BenOr 83**] M. Ben-Or, Another advantage of free choice: completely asynchronous agreement protocols (extended abstract), Proc. 2*nd* Ann. ACM Symp. Princ. Distr. Syst., Montreal (Aug. 1983) 27-30.

[**Berge 73**] C. Berge. Graphes et hypergraphes (Bordas, 1973).

[**Bernstein 80**] A.J. Bernstein, Output guards and nondeterminism in communicating sequential processes, ACM Trans. Prog. Lang. Syst. 2, 2 (1980) 234-238.

[**Best 84**] E. Best, Fairness and conspiracies, Inf. Proc. Letters 18 (1984) 215-220.

[**Best 85**] E. Best, Concurrent behavior: sequences, processes and axioms, Proc. CMU Workshop on Concurrency, Lect. Notes Comp. science 197 (1985) 221-245.

[**Bougé 84**] L. Bougé, Repeated synchronous snapshots and their implementation in CSP, Proc. 12*th* Int. Coll. Automata, Lang. and Progr., Nafplion, Greece, Lect. Notes Comp. Science 194 (Springer verlag, 1985). To appear in Theor. Comp. Science.

[**Bougé 85**] L. Bougé, Symmetry and genericity for CSP distributed systems, Rept. No. 85-32, LITP, Univ. Paris 7, Paris (May 1985). Submitted to Distributed Computing.

[**Bougé 86**] L. Bougé, On the existence of symmetric algorithms to find leaders in networks of Communicating Sequential Processes, Rept. No. 86-18, LITP, Univ. Paris 7, Paris (1986). To appear in Acta Informatica.

[**Burns 81**] J.E. Burns, Symmetry in systems of asynchronous processes, Proc. 22*nd* Symp. Found. Comp. Science, Nashville, Tennessee (1981) 169-174.

[**BBK 86**] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, Conditional axioms and α/β-calculus in process algebra, Rept. No. CS-R8502, Dept. Comp. Science, CWI, Amsterdam (Février 1985).

[**BC 84**] G. Berry, L. Cosserat, The Esterel synchronous language and its mathematical semantics, Rept. No. 327, INRIA, Reocquencourt, France (Sept. 1984).

[**BCKKM 83**] W. Bux, F. Closs, K. Kuemmerle, H. Keller, H.R. Mueller, The *Token Ring*, Local Area Network: an advanced course, Lect. Notes Comp. science 184 (Juillet 1983) 36-63.

[**BK 85**] R.J. Back, R. Kurki-Suonio, Serializability in distributed systems with handshaking, Rept. No. CMU-CS-85-109, Dept. Comp. Science, Carnegie-Mellon Univ., Pittsburgh, Penn. (1985).

[**BL 85**] H.L. Bodlaender, J. van Leeuwen, New upper bounds for decentralized extrema-finding in a ring of processors, Rept. No. RUU-CS-85-15, Dept. Comp. Science, Univ. Utrecht, The Netherlands (1985).

[**BS 83**] G.N. Buckley, A. Silberschatz, An effective implementation of the generalized input-output construct of CSP, ACM Trans. Prog. Lang. Syst. 5, 2 (1983) 223-235.

[**BT 83**] G. Bracha, S. Toueg, Resilient consensus protocols, Proc. 2*nd* Ann. ACM Symp. Princ. Distr. Syst., Montreal (Aug. 1983) 12-26.

[**Chandy 83**] K.M. Chandy, Paradigms for distributed computing, in: Proc. 3*rd* Conf. on Found. of Soft. Technology and Theory of Computer Science, Bengalore, India (1983) 192-201.

[**Charron 86**] B. Charron, Notion de symétrie dans le langage CCS; étude des systèmes électoraux symétriques, rapport de DEA, Dept. Math. Inf., Univ. Orléans (juin 1986).

[**CES 83**] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal specifications, a practical approach, Proc. 10*th* ACM Symp. Princ. Progr. Lang., Austin, Texas (1983).

[**CL 85**] K.M. Chandy and L. Lamport, Distributed snapshots: determining the global state of distributed systems, ACM Trans. Comp. Systems 3, 1 (1985) 63-75.

[**CLP 83**] S. Cohen, D. Lehmann, A. Pnueli, Symmetric and economical solutions to the mutual exclusion problem in a distributed system, Automata, Languages and Programming, 10*th* Coll., Barcelona, Spain, July 1983, Lect. Notes Comp. Science 154 (1983) 128-136.

[**CM 83**] K.M. Chandy, J. Misra, Preserving asymmetry by symmetric processes and distributed fair conflict resolution, Research Rept., Univ. Texas, Texas (1983).

[**CM 84**] K.M. Chandy, J. Misra, The drinking philosophers problem, ACM Trans. Prog. Lang. Syst. 6,4 (1984) 632-646.

[**CM 85**] K.M. Chandy and J. Misra, A paradigm for detecting quiescent properties of distributed computations, Rept. No. TR-85-02, Dept. Comp. Science, Univ. Texas, Austin, Texas (1985).

[**CM 85a**] K.M. Chandy and J. Misra, How processes learn (abstract), Dept. Comp. Science, Univ. Texas, Austin, Texas (1985).

[**Dijkstra 72**] E.W.G. Dijkstra, Hierarchical ordering of sequential processes, in: C.A.R. Hoare, R.H. Perrot, eds., Operaring Systems Techniques (Academic Press, 1972) 72-93.

[**Dijkstra 75**] E.W.G. Dijkstra, Guarded commands, non-determinacy and formal derivation of programs, Comm. ACM 18, 8 (1975) 453-457.

[**Dijkstra 83**] E.W.G. Dijkstra, The distributed snapshot algorithm of K.M. Chandy and L. Lamport, Letter No. EWD864a (1983).

[**DU 84**] P. Degano, U. Montanari, Distributed systems, partial ordering of events, and events structures, Proc. Int. Summer School on Control Flow and Data Flow - Concepts of Distr. Prog., Markoberdorf, Germany, July 1984, NATO ANSI Series F13 (Springer, 1985).

[**DFG 83**] E.W. Dijkstra, W.H.J. Feijen, A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, Inf. Proc. Letters 16 (1983) 217-219.

[**DM 86**] C. Dwork, Y. Moses, Knowledge and common knowledge in a byzantine environment: crash failure, Rept. No. MIT/LCS/TM-300, Lab. for Comp. Science, MIT, Cambridge, Mass. (July 1986).

[**DS 80**] E.W.G. Dijkstra, C.S. Scholten, Termination detection for diffusing computations, Inf. Proc. Letters 11 (1980) 1-4.

[**EF 82**] T. Elrad, N. Francez, Decomposition of distributed programs into communication-closed layers, Science Comp. Prog. 2 (1982) 155-173.

[**Fauconnier 84**] H. Fauconnier, Comportements infinis de programmes CSP, Rept. No. 84-45, LITP, Univ. Paris 7, Paris (1984).

[**Francez 80**] N. Francez, Distributed termination, ACM Trans. Prog. Lang. Syst. 2 (1980) 42-55.

[**Francez 86**] N. Francez, Fairness (Springer, 1986).

[**FHV 84**] R. Fagin, J.Y. Halpern, M.Y. Vardi, A model-theoritic analysis of knowledge: extended abstract, IBM Research Laboratory, San Jose, Calif. (1984).

[**FL 84**] G.N. Frederickson, N.A. Lynch, The impact of synchronous communication on the problem of electing a leader in a ring, Proc. 16*th* Ann. ACM Symp. Theory Comp. (1984) 493-503.

[**FR 82**] N. Francez and M. Rodeh, Achieving distributed termination without freezing, IEEE Trans. Soft. Eng. SE-8 (1982) 287-292.

[**FRS 81**] N. Francez, M. Rodeh, M. Sintzoff, Distributed termination with interval assertion, Proc. Int. Coll. on Formalization of Programming Concepts, Peniscola, Spain, Lect. Notes in Comp. Science 107 (1981).

[**Gafni 86**] E. Gafni, Perspectives on distributed networks protocols: a case for building blocks, Proc. MILCOM'86, Monterey, Calif. (Oct. 1986).

[**Garcia 82**] H. Garcia-Molina, Elections in a distributed computing system, IEEE Trans. Computers C31, 1 (1982) 48-59.

[**Gastin 86**] P. Gastin, A model without global time for distributed systems, Rept. No. 86-12, LITP, Univ. Paris 7, Paris (1986).

[**Gouda 81**] M.G. Gouda, Distributed state exploration for protocol validation, Rept. No. 185, Dept. Computer Science, Univ. Texas, Austin, Texas (1981).

[**GA 84**] E. Gafni, Y. Afek, Election and traversal in unidirectional networks, Proc. 3*rd* Ann. ACM Symp. Princ. Distr. Comp., Vancouver (Août 1984) 190-198.

[**GAK 84**] E. Gafni, Y. Afek, L. Kleinrock, Fast and message-optimal synchronous election algorithm for complete networks, Rept. No. CSD-840041, Comp. Science. Dept., Univ. Calif., Los Angeles (1984).

[**GN 84**] A. Goyal, R. Nelson, Distributed election in local ring network, Rept. No. RC 10495, IBM T.J. Watson Research Center, Yorktown Heights, N.Y. (1984).

[**GS 86**] R. Gerth, L. Shrira, On proving communication closedness of distributed layers, proc. 5*th* Conf. on Found. Syst. Theory and Theor. Comp. Science, New-Delhi, Lect. Notes Comp. Science (Springer, 1986).

[**GL 85**] U. Gotz, R. Loogen, Towards a non-interleaving semantic model for CSP-like languages, Lehrstuhl fur Informatik II, RWTH, Aachen, Germany (March 1985).

[**Hoare 78**] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (1978) 666-677.

[**Hoare 84**] C.A.R Hoare, programs are predicates, Math. Logics and Progr. Lang., Phil. Proc. Royal Society London A (Feb. 1984) 141-154.

[**HBR 84**] C.A.R. Hoare, S.D. Brookes, A.W. Roscoe, A theory of communicating sequential processes, Journ. ACM 31 (1984) 560-599.

[**HM 84**] J.Y. Halpern, Y. Moses, Knowledge and common knowledge in a distributed environment, Proc. 3*rd* Ann. Symp. Princ. Distr. Comp., Vancouver, B.C., Canada (1984) 50-61.

[**HMR 85**] J.M. Helary, A. Maddi, M. Raynal, Contrôler les transferts de connaissance dans les algorithmes distribués; application à la détection de l'interblocage, Rept. No. 260, IRISA, Rennes (Juin 1986).

[**HO 85**] C.A.R. Hoare, E.-R. Olderog, Specification-oriented semantics (revised version), Rept. No. 8506, Dept. Comp. Science, Univ. Kiel (1985).

[**Inmos 84**] Inmos Limited, Occam programming manual (Prentice-Hall, 1984).

[**IR 81**] A. Itai, M. Rodeh, Symmetry breaking in distributive networks, Proc. 22nd Symp. Found. Comp. Science, Nashville, Tennessee, (1981) 169-174.

[**JS 85**] R.E. Johnson, F.B. Schneider, Symmetry and similarity in distributed systems (extended abstract), Comp. Science Dept., Cornell Univ., Ithaca, New York (Feb. 1985).

[**KMZ 84**] E. Korach, S. Moran, S. Zask, Tight lower bounds and upper bounds to some distributed algorithms for a complete network of processors, Proc. 3*rd* Ann. ACM Symp. on Distr. Comp., Vancouver, B.C., Canada (Aug. 1984) 199-207.

[**KKM 86**] E. Korach, S. Kutten, S. Moran, A modular technique for the design of efficient leader finding algorithms, Proc. 4*th* Ann. ACM Symp. Princ. Distr. Comp., Minaki, Ontario, Canada (Aug. 1985), 163-175.

[**KRS 83**] E. Korach, D. Rotem, N. Santoro, Distributed election without a global sense of orientation, Rept. No. SCS-TR-17, School of Computer Science, Carleton Univ. (Jan. 1983).

[**KT 86**] S. Katz, G. Taubenfeld, What processes know: definitions and proof system, Proc. 5*th* Ann. ACM Symp. Princ. Distr. Comp., Calgary, Canada (Août 1986) 249-262.

[**Lamport 78**] L. Lamport, Time, clocks and the ordering of events in a distributed system, Comm. ACM 21, 7 (1978) 558-564.

[**Lamport 84**] L. Lamport, Lecture Notes prepared for the Advanced Course on Distributed Systems - Methods and Tools for Specification, Computer Science Inst., TUM, Munchen, Germany (1984).

[**Lehmann 84**] D. Lehmann, Knowledge, common knowledge and related puzzles (extended summary), Proc. 3*rd* Ann Symp. Princ. Distr. Comp., Vancouver, B.C., Canada (1984) 62-67.

[**LeLann 77**] G. Le Lann, Distributed systems - towards a formal approach, in: B. Gilchrist, ed., Information Processing 77 (North-Holland, Amsterdam, 1977) 155-160.

[**LeVerrand 82**] D. Le Verrand, Ada, manuel d'évaluation (Dunod, 1982).

[**LPS 81**] D. Lehmann, A. Pnueli, J. Stavi, Impartiality, justice and fairness: the ethics of concurrent termination, Lect. Notes Comp. Science 115 (1981) 264-277.

[**LR 81**] D. Lehmann, M.O. Rabin, On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem (extended abstract), Proc. 8*th* Ann. ACM Symp. Princ. of Prog. Lang., Williamsburg, Virginia (Jan. 1984) 133-138.

[**LS 84**] L. Lamport, F. Schneider, CSP Hoare's logic and all that, ACM Trans. Progr. Lang. Systems 6,2 (1984) 281-296.

[**LSP 82**] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Trans. Princ. Progr. Lang. 4, 3 (1982) 382-410.

[**Milner 80**] R. Milner, A Calculus for communicating systems, Lect. Notes Comp. Science 92 (Springer, 1980).

[**Merritt 84**] M. Merritt, Elections in the presence of faults, Proc. 3*rd* Ann. ACM Symp. Princ. Distr. Comp., Vancouver, B.C., Canada (1984) 134-142.

[**Morgan 85**] C. Morgan, Global and logical time in distributed systems, Inf. Proc. Letters (1985), to appear.

[**MC 82**] J. Misra and K.M. Chandy, Termination detection of diffusing computations in communicating sequential processes, ACM Trans. Prog. Lang. Syst. 4 (1982) 37-42.

[**MT 86**] Y. Moses, M. Tuttle, Programming simultaneous actions using common knowledge: preliminary version, Proc. ACM Symp. Found. Comp. Science (1986).

[**OA 86**] E.-R. Olderog, K. Apt, Fairness in parallel programs: the transformational approach (revised version), Rept. No. 8611, Dept. Comp. Science, Univ. Kiel (1986).

[**OH 85**] E.-R. Olderog, C.A.R. Hoare, Specification oriented semantics for communicating processes (revised version), Rept. no. 85-06, Dept. Comp. Science, C. Albrecht Univ., Kiel (Oct. 1985).

[**Plotkin 82**] G. Plotkin, An operational semantics for CSP, in: D. Bjorner, ed., Formal Description of Programming Concepts, IFIP TC-2 Working Conference, Garmish-Partenkirchen, Germany, 1982, (North-Holland, 1983) 199-223.

[**QH 86**] J.S. Quaterman, J.C. Hoskins, Notable computer networks, CACM 29, 10 (Octobre 1986) 932-971.

[**Rana 83**] S.P. Rana, A distributed solution to the distributed termination problem, Inf. Proc. Letters 17 (1983) 43-46.

[**Reisig 84**] W. Reisig, Partial order semantics for CSP-like languages and its impact on fairness, Automata, Languages and Programming, 11*th* Coll., Antwerpen, Belgium, 1984, Lect. Notes Comp. Science 172 (1984) 403-413.

[**Reisig 85**] W. Reisig, A minimal order semantics for CSP, GMD, Sankt Augustin, Germany (Février 1985).

[**Richier 83**] J.-L. Richier, Un sur-langage CSP de Pascal, Rept. No. 83-24, LITP, Univ. Paris 7, Paris (1983).

[**Richier 84**] J.-L. Richier, Distributed termination in CSP - symmetric solutions with minimal storage, Rept. No. 84-49, LITP, Univ. Paris 7, Paris (1984).

[**Roscoe 84**] A.W. Roscoe, A denotational semantics of OCCAM, Proc. CMU Workshop on Concurrency, Lect. Notes Comp. science 197 (1985).

[**SDRC 83**] J.F. Shoh, Y.K. Dalal, D.D. Redell, R.C. Crane, The *Ethernet*, Local Area Network: an advanced course, Lect. Notes Comp. science 184 (Juillet 1983) 1-35.

[**SF 86**] N. Shavit and N. Francez, Efficient detection of locally-indicative stable properties, Proc. Int. Coll. Automata, Languages and Prog., Rennes, 1986, Lect. Notes Comp. Science 194 (Springer, 1986).

[**SK 82**] N. Santoro, R. Khatib, Routing without routing tables, Rept. No. SCS-TR-6, School Comp. Science, Carleton Univ., Ottawa (July 1982).

[**TL 86**] R.B. Tan, J. van Leeuwen, General symmetric distributed termination detection, Rept. No. RUU-CS-86-2, Dept. Comp. Science, Rijksuniversitet, Utrecht, Pays-Bas (1986).

[**TPS 84**] S. Toueg, K.J. Perry, T.K. Srikanth, Fast distributed agreement, Rept. No. TR84-621, Dept. comp. Science, Cornell Univ., Ithaca, NY (July 1984).

[**Vitanyi 84**] P.M.B. Vitanyi, Distributed elections in an archimedian ring of processors, Proc. 16*th* Ann. ACM Symp. Theory of Comp. (1984) 542-547.

[**WS 83**] D.D. Wright, F.B. Schneider, A distributed path algorithm and its correctness proof, Rept. No. TR 83-556, Dept. of Computer Science, Cornell Univ., Ithaca, New York (1983).

Annexe 1: Un conte de terminaison répartie

d'après Friedemann Mattern, Université de Kaiserslautern, R.F.A.

Il était une fois* ...le roi de Polymocronie. Il se dit un jour que l'heure était venue de partager équitablement son royaume aux îles innombrables entre ses petits-enfants. Il envoya aux sages répartis dans tout son royaume des messagers afin qu'ils lui donnent leur avis.

Le roi connaissait bien l'indépendance et la constance de ses conseillers, qui ne faisaient que manger et penser tout au long du jour. Que l'un d'eux soit en train de manger ou de boire, seul un émissaire royal ou un message d'un autre membre du conseil pouvait l'amener à penser de nouveau. Il se saisissait alors immédiatement d'une plume, de papier, d'encre et du sceau pour envoyer un nouvel avis à d'autres membres de ce conseil royal tellement éparpillé. Affamé par un tel effort, il s'en retournait bientôt à la table, toujours chargée de mets princiers.

Les années passèrent, et le roi se faisait de plus en plus vieux. Il n'avait jamais reçu une seule réponse à sa demande de la part de ses sages. Il soupira, et s'adressa à son conseiller secret en ces termes: "Je sais combien il est difficile de partager mon domaine avec justice. Je connais les règles de mon conseil, qui ne rend son avis que lorsque le problème a été suffisamment débattu pour que chacun des sages soit satisfait et qu'aucun messager ne soit encore en chemin. Il n'y a que les PTT royaux qui m'inquiètent. Et si l'un des navires avait sombré dans la tempête ou s'était perdu aux confins du royaume?"

"Majesté royale", répondit le conseiller secret, "votre royaume est immense, et les marins mettent souvent longtemps pour aller d'une île à l'autre. Personne ne peut prévoir combien de temps le voyage prendra, et il arrive que, pendant la nuit, un navire en double un autre. Mais le talent des marins, l' Ecole Normale Supérieure (de la marine) qui les forme et le sens du devoir de vos serviteurs garantissent qu'aucun message ne se perd. Envoyons donc des messagers, Majesté, pour demander à chaque conseiller combien de messages il a reçus et combien il en a envoyés. Alors, nous pourrons facilement en déduire si, *summa summarum*, autant de messages ont été reçus qu'envoyés, et donc si le conseil royal a tranché.

Le roi fut très heureux de ces paroles avisées. Il reprit confiance, et demanda immédiatement à son Grand Mathématicien de concevoir un plan. Celui-ci se présenta avec une carte, et dit: "Majesté, vous voyez sur ce scénario que l'idée de votre conseiller peut conduire à des conclusions erronées. Les messagers envoyés aux îles A et B rapportent en effet qu'il y a autant de messages reçus que de messages envoyés, *summa summarum*, exactement un message. Mais il y a encore des messages en route! le problème est que les messagers doivent progresser avec précaution et ne pas se laisser tromper, pour évaluer correctement le compte royal. En effet, *primo* nous n'avons pas encore découvert l'horloge, et donc nous ne pouvons déterminer une heure commune. *Secundo*, nous ne connaissons pas encore la radio-diffusion. *Tertio*, la coutume immémoriale défend que les sages se rasssemblent en un même lieu. Le roi fut troublé de ces paroles étranges, et il dit: "Je le sais bien, puisque je suis le roi. Et que me conseillez-vous?" "Majesté,mon plan prévoit d'envoyer de nouveau les messagers aussitôt que le dernier d'entre eux est rentré au palais. Si les résultats obtenus sont identiques, les sommes étant égales, alors il n'y a plus aucun message en route." "Excellent!" dit le Roi qui n'avait rien compris. "Occupez-vous donc immédiatement de donner aux messagers leurs instructions!"

Le Grand Mathématicien fit ainsi, améliora encore son plan et voici bientôt que les messagers, investis de toute la puissance royale, firent voile sur les meilleurs vaisseaux du royaume vers les sages.

Quand finalement la décision des sages parvint au palais, le roi était si heureux qu'il éleva son Grand Mathématicien à la dignité de Grand Informaticien. et celui-ci recherche encore, s'il n'est pas déjà mort, dans une tour isolée du palais ... une meilleure solution au problème de la terminaison répartie!

---

* L'original de ce texte en allemand est paru dans Informatik-Spektrum (Springer Verlag, 1985) 8: 342-343. Je remercie tous ceux (et celles!) qui m'ont aidé à traduire ce texte.

Note du conteur

Mais s'il est déjà mort, ce n'est pas si tragique, puisqu'il éduquait de bons disciples qui ont poursuivi sa tâche. Ceux qui voudraient en savoir plus sur ce problème important qu'est la terminaison répartie, et sur les solutions successives qui ont été proposées, peuvent s'adresser à

Dipl.-Inform. F. Mattern
Univ. Kaiserslautern
Sonderforschungsbereich 124
"VLSI-Entwurfsmethoden und parallelität"
Postfachen 3049
D-6750 Kaiserslautern

et aussi à

Luc Bougé
Laboratoire d'Informatique
Ecole Normale Supérieure
45, rue d'Ulm
75230 Paris Cédex 05
`inria!ens!bouge` ou `bouge@frulm63.bitnet`
et
Laboratoire d'Informatique
Université d'Orléans
BP 6759
45067 orléans Cédex 02
`inria!univorl!bouge`

Epilogue

Le papier valait tellement cher à cette époque que le Grand Informaticien renonça à écrire la preuve de la correction de son plan. Nous ne possédons que quelques notes qui montrent qu'il imagina plus tard un procédé où chaque île n'est pas nécessairement visitée deux fois ou plus. Il parvint donc manifestement à majorer ainsi le nombre des messagers nécessaires par une fonction linéaire du nombre d'îles et de messages. Malheureusement, la méthode n'est pas décrite. Qui pourrait aider à la retrouver et à la vérifier?