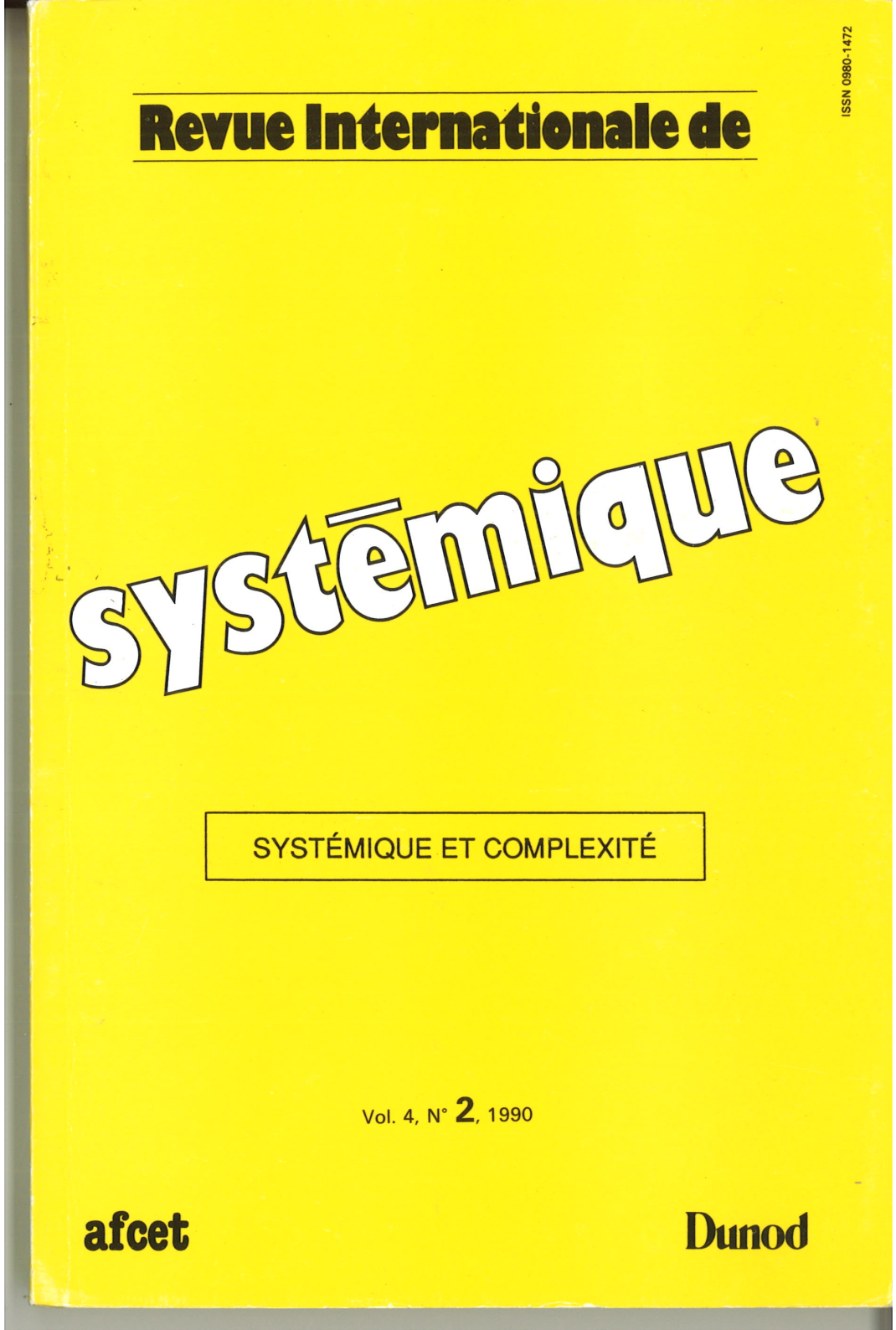

AFSCET

Revue Internationale de systémique

Revue
Internationale
de Sytémique

volume 04, numéro 2, pages 189 - 201, 1990

Essai sur la complexité des systèmes informatiques

Edmond Bianco

Numérisation Afscet, janvier 2016.

REVUE INTERNATIONALE DE SYSTÉMIQUE
Vol. 4, N° 2, 1990, pp. 189 à 201

ESSAI SUR LA COMPLEXITÉ DES SYTÈMES INFORMATIQUES

Edmond BIANCO
Université d'Aix-Marseille II [1]

Classe générale des systèmes

Un système informatique présente deux aspects : vu de l'extérieur, il est apprécié par l'utilisateur selon sa discrétion et son efficacité. Moins le système est visible meilleur c'est, dans la mesure où il assure à notre utilisateur un service complet qui lui permet de retravailler à sa guise programmes et configurations.

Vu de l'intérieur la structure en est d'autant plus complexe que le service doit être de bonne qualité.

Pour la commodité de l'exposé je définis quelques notions indispensables. Je me place dans la situation où l'utilisateur standard attend du système devant lequel il est assis, la plus grande variété possible de services. Il doit pouvoir faire appel à n'importe quel langage de programmation dont il a besoin, et pouvoir disposer de toutes les manipulations indispensables pour assurer aisément sa réalisation.

Je désigne par tâche le travail qu'il entame, assis devant son terminal. Il réclame donc le langage qu'il désire utiliser. A partir de cet instant il doit pouvoir disposer de tout ce qui est nécessaire pour traiter les programmes mais aussi des données qui constituent la trame du calcul. Tout cela doit pouvoir être construit, relu, vérifié, modifié, corrigé, à tout instant et à la demande.

Or, c'est ce qui constitue le cadre spécifique de la tâche entamée, et le genre de traitement qui lui correspond dépend en toute généralité du langage utilisé. Dans le système qui comporte le degré maximum de généralité puisqu'il doit gérer des tâches traitées dans des langages divers, je désigne par sous-système l'ensemble des services qui peuvent être rendus à propos

1 Faculté de Sciences Economiques, L.I.T.A.M., rue Puvis de Chavannes, 13013 Marseille.

Rev. intern. systémique. 0980-1472 /90/02/189 - 13/$ 3.30

de l'utilisation d'un langage. Le sous-système est de degré de généralité évidemment moindre.

Le sous-système est constitué de trois langages : le langage de programmation, le langage d'échanges qui permet de constituer et de manipuler des configurations de données, et un langage de communication dont les objets sont programmes et configurations.

Avec tout cela on peut construire des programmes, préparer des configurations de données, appliquer les programmes sur les configurations, puis manipuler programmes et configurations, stockage, destruction, correction, enfin traiter les résultats.

Selon le degré d'ergonomie atteint on peut envisager des déroulements plus ou moins contrôlés de programmes, avec des traces plus ou moins serrées. On peut également visualiser les données sous les formes le plus élaborées, à partir de langages d'échanges très performants.

Création d'une tâche

C'est le scénario le plus simple possible qui préside à l'élaboration d'une tâche. L'utilisateur s'installe devant son terminal, il dispose d'un langage, je dirai le langage hors-texte, qui lui permet de choisir le sous-système dont il a besoin.

A partir du moment où le choix est fait, l'utilisateur dispose du langage de communication. Et c'est dans ce nouveau cadre qu'il va pouvoir exercer d'autres choix spécifiques. Il peut demander à travailler avec le langage de programmation, ou bien avec le langage d'échanges pour constituer une configuration, ou encore lister des données, et chaque travail terminé renvoit dans le langage de communication, c'est de là qu'on peut lancer un programme en l'appliquant sur une configuration.

En cours de travail, l'utilisateur peut demander à suspendre une tâche pour pouvoir la reprendre plus tard. C'est le langage de communication qui le lui permet.

Ce qui vient d'être dit est valable quel que puisse être le nombre d'utilisateurs susceptibles d'agir simultanément. Aussi bien peut-il n'y en avoir qu'un seul ce qui, contrairement à ce qu'on pourrait penser, ne réduit pas la complexité du système. En effet, un utilisateur peut demander à dérouler un ou plusieurs programmes pendant que lui-même en compile un autre tranquillement. Même les plus petits ordinateurs sont désormais assez puissants pour réaliser ce genre de calculs.

Structure interne de systèmes

La notion de système telle que je l'envisage, doit comporter une propriété fondamentale : l'ensemble des sous-systèmes qu'un système gère est extensible.

Cette propriété a des conséquences importantes sur la structure du système lui-même qui ne peut ainsi avoir avec les sous-systèmes qu'il gère à un instant donné, que des relations qui en assurent l'indépendance mutuelle.

Si je construits un système en même temps que les compilateurs dont il contrôle le fonctionnement, je peux me permettre de le façonner pour leur faciliter le travail, au moins dans une certaine mesure. Un tel système adapté à un sous-système spécifique, risque de l'être moins pour un sous-système un tant soit peu différent. De plus un sous-système rajouté après coup est en principe parfaitement inconnu au moment de la construction du système, qui ne peut donc pas tenir compte de ses particularités.

Le problème est ainsi clairement posé : quelle est donc la nature de relations qui doivent exister entre un système et un jeu à la fois quelconque et extensible de sous-systèmes ?

Pour répondre à une telle question il faut se donner quelques moyens. D'abord, séparer nettement les responsabilités. Le système, on l'a vu, assure la gestion de l'ensemble du point de vue de la relation au fini illimité. Si l'on ajoute à cela que plusieurs tâches vont avoir à se dérouler en parallèle, alors on est amenés à introduire la notion d'autojectivité [1].

Si la précaution est prise de rendre autojectifs tous les compilateurs qui composent un sous-système, alors les rapports qui existent entre le système et chaque compilateur sont réduits au minimum de complexité.

Bien entendu l'autojectivité a des implications sur une structure de sous-système. Par principe, l'utilisateur qui désire employer les services d'un sous-système quelconque fera généralement appel successivement à plusieurs sortes de traitements. Chaque traitement met en œuvre un compilateur du sous-système, ainsi plusieurs compilateurs devront se succéder. L'autojectivité interdit que deux compilateurs s'insèrent l'un dans l'autre. Le transfert d'insertion ne peut que passer par le système.

L'intérêt du sous-système réside dans l'indépendance qu'il est censé donner à l'utilisateur pour réaliser son travail. Par exemple, je veux compiler un programme, puis le sauvegarder sur disquette, constituer une configuration pour l'essayer, et enfin appliquer mon programme sur cette

configuration. Donc, dans l'ordre j'aurai affaire, à l'intérieur de mon sous-système à une succession des divers compilateurs qui le composent.

Ainsi si je désigne par Clc et Cdlc, les compilateurs traducteur et dérouleur du langage de communication, par Cle et Cdle les compilateurs du langage d'échange, et Clp et Cdlp ceux du langage de programmation, pour le problème posé je vais avoir affaire avec la succession :

Clc, Cdlc (choix du Lp), Clp (compilation du P), Clc, Cdlc (sauvegarde P), Clc, Cdlc (choix du langage d'échange), Cle (compilation de la configuration), Clc, Cdlc (application du P sur la configuration).

Ces compilateurs vont donc se succéder les uns aux autres au gré du travail qui leur est demandé. Pour conserver un contrôle cohérent du déroulement il est indispensable que l'enchaînement se fasse dans le système, seule condition pour la conservation de l'autojectivité.

Il devient alors nécessaire de se donner les moyens de transférer l'information d'un compilateur quelconque, qui définit son successeur en fonction des circonstances, vers le système qui va insérer ce successeur, en lui passant les paramètres. C'est la notion de vecteur information d'insertion fractionnée qui permet de maîtriser la difficulté.

Le seul organe qui puisse ainsi assurer la liaison entre un système à structure définitive et un compilateur appartenant à un sous-système apparu ultérieurement ne peut être que la machine universelle. Car c'est elle qui déroule alternativement l'un et l'autre.

Le vecteur information est une configuration commune au système et à la machine universelle dont le contenu permet la communication entre système et compilateur par l'intermédiaire de cette machine universelle. Afin de simplifier les choses, il suffit de disposer d'une instruction spéciale, utilisable dans le compilateur, dans laquelle on précise le nom du successeur, et les files qui contiennent l'information à transmettre.

Du point de vue algorithmique le mécanisme est le suivant :

dans le système l'instruction d'insertion fractionnée fournit en paramètre l'accès à un vecteur d'information qui doit contenir les caractéristiques du compilateur à insérer. La machine universelle qui déroule cette instruction consulte ce vecteur information et en conserve l'accès, puisque par construction elle ne pourra rien dérouler d'autre que le compilateur inséré jusqu'à ce que le chemin fini borné de celui-ci soit achevé.

Là, deux cas peuvent se produire, une fin normale sur la rencontre de la sortie du chemin déroulé renvoit simplement dans le système à l'instruction qui suit l'insertion fractionnée. La rencontre dans le chemin d'une instruction d'arrêt oblige la machine universelle à noter ses paramètres dans le vecteur information, en l'occurrence le nom du compilateur successeur, et les accès aux files paramètres. Le retour se fait également à la ligne qui suit l'insertion fractionnée, à charge pour le système d'explorer le vecteur information pour y trouver le nom du compilateur successeur qui viendra en paramètre à la prochaine insertion.

Pour lui faciliter ce travail, le vecteur information contient une variable d'état qu'il suffit de consulter pour savoir s'il faut ou non changer de compilateur. Cet état est modifié par la machine universelle en particulier sur la rencontre de l'instruction d'arrêt, il est initialisé et recalculé par le système.

Outre la notion habituelle d'instruction qui est le moyen normal au sens de Von Neumann de construction d'algorithme pour la machine universelle, on voit apparaître une notion de tableau de paramètres destiné à des échanges directs d'information entre programme et machine universelle. Ceci se révèle indispensable pour assurer le respect de la notion d'autojectivité. Laquelle est elle-même indispensable pour assurer l'indépendance système-sous-système.

Notions et concrétisation de notions

Après avoir ainsi survolé rapidement une structure de système en soulignant les points durs, les endroits de l'algorithme où se concentrent certaines difficultés, je vais préciser un peu les moyens.

Le problème à résoudre met en jeu, donc, à la fois compilateurs et systèmes. Je ne me préoccupe pas de la compilation, mais seulement de l'aspect du compilateur vu de l'extérieur, en particulier vu du système. Il nous suffit de savoir que tout compilateur peut être mis sous forme autojective.

Résoudre le problème posé, c'est-à-dire se donner les moyens de décrire entièrement et de manière effectivement calculable un système doué des propriétés énoncées, revient à construire un langage de programmation. Pour que le but soit atteint il est nécessaire que toutes les parties délicates de l'organisation du système puissent être rassemblées sous des formes rigides et simples telles qu'on s'attend à les trouver dans les langages.

Il faut de plus séparer ce qui doit continuer à se programmer parce qu'on a affaire à des formes évolutives dépendant de cas singuliers, de ce

qui doit se retrouver figé dans des formes définitives car cela recouvre des notions indépendantes des circonstances. Toutefois pour les formes évolutives des moyens appropriés doivent être prévus.

Comme toujours en matière de langages, bien qu'étroitement mêlés il est commode de dissocier aspects algorithmiques et aspects structures de données. Je vais commencer par les éléments essentiels de l'algorithme.

Bien entendu condenser une opération compliquée sous une forme simple ne gomme en aucune manière la difficulté mais en fait la déplace. La charge de programmation normalement dévolue à l'utilisateur se retrouve dans la machine universelle. Le gain réside dans le fait que le travail est fait une fois pour toutes. S'il existe un léger surcoût dû au fait que la fonction doit être générale donc un peu plus complexe, il existe une énorme économie due à la facilité d'analyse apportée ainsi à l'utilisateur. Mais bien sur, le prix à payer est celui de la culture à acquérir pour pouvoir utiliser de tels moyens. Quant on constate que nous commençons à peine à utiliser cette vieillerie de langage C, on peut être tenté de penser que la culture coûte fort cher, et donc n'est pas à la portée de tout le monde.

Comme on vient de le voir la notion fondamentale autour de laquelle tout est construit est l'autojectivité. C'est elle qui rend le système totalement indépendant de ce qui fait la spécificité du travail du compilateur.

Pour insérer un compilateur le système dispose d'une instruction spécifique : l'insertion fractionnée. Et c'est autour de cette instruction qu'est organisée la gestion des compilateurs. Dans cette gestion, une partie doit demeurer souple donc programmable c'est ce qui assure la particularité du système. Par contre, et c'est ce qui nous intéresse ici, c'est le jeu lui-même de l'insertion avec ses flux d'information qui est universel.

Je pars des propriétés suivantes :

Le sous-système est autonome dans son déroulement par rapport au service qu'il doit rendre, donc ce sont les divers compilateurs qui le composent qui doivent désigner leur successeur en fonction de l'information qu'ils ont collecté.

Deux compilateurs ne peuvent se succéder directement sans passer par les systèmes sous peine de détruire l'autojectivité.

L'information ne peut passer directement du compilateur au système puisque lors de la construction de ce dernier le sous-système n'existait pas forcément déjà.

En conclusion ce n'est que par l'intermédiaire de la machine universelle que l'information peut transiter. C'est le rôle dévolu au vecteur information de l'insertion fractionnée que de servir de boite aux lettres. D'autant qu'entre le moment de l'insertion et celui du retour, par définition autojective, la machine universelle ne peut dérouler qu'un seul chemin appartenant à un seul compilateur.

Cela assure le bon déroulement des opérations au moyens de trois éléments du langage :

L'instruction d'insertion, l'instruction d'arrêt, le vecteur information.

Le premier est localisé exclusivement dans le système, le second dans le compilateur où il permet de préciser le nom du compilateur successeur et la liste des files paramètres, le troisième présente une structure déterminée par le langage car, s'il est programmable à la demande selon les particularités de chaque système, il est également utilisé par la machine universelle.

Une question persiste : tout compilateur peut-il être mis sous forme autojective ? Il existe en effet nombre d'opérations de compilation qui ne sont pas finies-bornées. Déjà une phrase à compiler n'est a priori pas finie-bornée. Constituer des tables d'identificateurs et les relire, construire un généré, ne sont pas des opérations dont on peut à l'avance prévoir la longueur.

C'est la notion de boucle à ouverture qui permet d'absorber cette difficulté. Par construction seules les boucles à argument calculé peuvent n'être pas bornées à l'avance. Il suffit donc dans le langage d'éviter les boucles intempestives. Toute commutation aval se réalise sous la rubrique aiguillage, il n'est pas de commutation amont en dehors de la notion de boucle qui se trouve alors parfaitement contrôlée si on la munit d'une ouverture [2].

C'est la maîtrise de la structure de l'objet qui représente l'autre aspect de la question. En effet tout objet traité est codé en mémoire, le code est le résultat du choix d'une représentation, et rien ne garantit a priori que la variation de cette représentation demeure finie-bornée au cours de l'évolution des calculs.

Il est donc nécessaire dans le langage lui-même de contrôler la géométrie de la représentation. La notion utilisée à cet effet est la notion de statut. A toute variable est attaché un statut qui définit sa représentation sur un système de référence défini lui aussi dans le langage. Lorsqu'au cours d'un calcul une représentation déborde, un signal est émis par la machine universelle.

Dans un même programme un statut est une constante déclarée. Mais un statut peut devenir une variable dans le cadre limité suivant : au cours

d'un calcul on tombe sur un élément dont on ignore ou dont on a oublié le statut, il faut donc pouvoir le récupérer afin de le replacer dans l'un des seuls choix possibles, une instruction spéciale permet ce calcul.

Il existe une autre version de la notion de statut mais destinée à l'usage exclusif du système. Il s'agit du statut global qui lui permet de traiter en bloc des paquets d'informations dont il n'a pas à connaître le détail intérieur mais dont il doit connaître les caractéristiques extérieures pour pouvoir en assurer la gestion de l'encombrement.

Au niveau supérieur d'organisation, les éléments sont accumulés dans des files qui sont globalement finies-bornées, et dont la structure est destinée à faciliter le calcul.

Enfin une structure de file spéciale, la file support donne le moyen au système de traiter en bloc les images réelles des diverses files qui appartiennent aux compilateurs.

Organisation des Systèmes

Sans avoir le même degré de complexité des grands systèmes économiques ou autres, un tel système informatique est loin d'être une construction simple, d'autant que sa structure dépend en fait de nombre de facteurs psychologiques. Et le Dieu des informaticiens sait si les Utilisateurs d'informatique ont une psychologie bizarre.

J'ai essayé de montrer comment décrire entièrement une telle organisation à l'aide d'un système linguistique cohérent. A partir du moment où la description, complète, ne laisse aucune zone d'ombre dans laquelle pourraient se cacher de nombreux et dangereux pièges sémantiques, il devient possible de raisonner avec moins de chances de désagréables surprises.

Je vais donc tenter une étude de répartition du degré d'organisation. Pour cela je vais poser quelques définitions qui me paraissent commodes.

Je dirai qu'une tâche déterminée est dotée d'un degré constant d'organisation tant qu'on reste dans le même domaine de travail, ainsi quand j'utilise le compilateur d'un langage, et tant que je travaille sous son contrôle, je ne change pas de degré. A partir du moment où je désire changer de compilateur j'engage une procédure qui me fait sortir du langage que j'utilise pour pénétrer dans un autre différent où les problèmes d'organisation sont aussi différents.

D'abord je dirai que tous les actes et opérations qui appartiennent à un même ensemble de traitements de compilation constituent un plan d'organisation. Compilation est pris ici au sens le plus large, qu'elle soit de la traduction ou du déroulement.

Ainsi imaginons un utilisateur en train de compiler un programme, le travail du compilateur auquel il fait appel se situe donc sur un même plan d'organisation qui est lié à la nature du langage compilé. Mais en même temps, le traitement se déroule simplement parce qu'une machine universelle déroule le code du compilateur. Le travail de cette machine se situe donc sur un autre plan d'organisation qui est lié, lui, au langage qui a servi à construire le compilateur. On constate que ces deux plans coexistent dans un même intervalle de temps. Mais la machine outre le compilateur, et alternativement, déroule également ce programme qu'on appelle le système.

Je situe naturellement les opérations système sur un même plan d'organisation. L'organisation totale d'une opération de compilation se décompose donc en trois plans d'organisation. A cela il va falloir ajouter les relations entre ces divers plans.

Je peux définir un canal qui permettrait la communication entre deux plans, mais surtout ce qui est plus important, c'est un point de communication entre plusieurs plans, voire entre tous les plans : le nœud d'organisation.

Or, dans le système, précisément les compilateurs ne peuvent chacun communiquer qu'avec lui. Si dans un sous-système, deux compilateurs ont à échanger de l'information il leur faut passer par l'intermédiaire du système. C'est là que se situe le nœud de l'organisation.

L'insertion fractionnée avec ses mécanismes complémentaires, vecteur information et instruction d'arrêt, en est la matérialisation.

En quelque sorte on peut ainsi détailler la progression dans les changements de degré d'organisation pour la classe générale de nos systèmes. Quand je travaille dans un plan d'organisation, il me faut disposer d'un indicateur sémantique de changement. Autrement dit chaque langage doit offrir au moins un moyen linguistique de dire qu'on arrête, ne serait-ce que la lettre fin qui ponctue toute phrase. Le compilateur qui rencontre cette indication compose un message qui sera lu par le système. Ce dernier qui renvoyait autojectivement sur le même compilateur pour le compte de la même tâche, renvoit au coup suivant sur le nouveau compilateur demandé. Et renverra désormais autojectivement sur ce nouveau compilateur jusqu'à ce que celui-ci lui demande un changement. Et ainsi de suite. La façon dont les compilateurs se renvoient les uns aux autres constitue une organisation d'un degré plus élevé qui est celle d'un sous-système. Cette organi-

sation est, bien entendu, à la charge du concepteur du sous-système, et elle doit déjà se trouver explicitement symbolisée dans les langages qui composent cette organisation.

On constate donc qu'il existe une organisation d'ordre symbolique sur le plan abstrait de la construction des langages de programmation, en dehors de tout support de déroulement mécanique. Et qu'il existe une organisation sous-jacente, basée sur l'autojectivité, indépendante de chaque tâche qui peut être mise en œuvre dans le cadre de chaque sous-système, mais qui en permet le déroulement selon des normes acceptables du point de vue de la gestion du matériel et du service rendu.

Si je considère alors l'organisation qui règne sur le déroulement de l'ensemble de ces programmes constitué par tous les compilateurs actifs et le système, deux situations se présentent alors. D'abord tout naturellement le système est un programme de même nature que les compilateurs, comme eux il est enregistré en mémoire sous forme de code.

Cela signifie qu'un compilateur dérouleur spécial, l'unité centrale ou processeur, comme on veut, va se charger de lire ces codes pour les dérouler.

C'est finalement cet organe qui assure l'enchaînement effectif d'un plan sur l'autre, passant alternativement du code-système au code-compilateur et retour dans un espace pré-structuré.

Ensuite une telle organisation d'ensemble est conçue précisément pour que ce programme appelé système soit doté d'une structure définitive indépendante des sous-systèmes qu'on pourra lui confier successivement au cours de l'évolution de l'ensemble. Ceci a pour conséquence directe que le système peut être intégré au même niveau que le processeur. Cela signifie que des échanges directs sont réalisables entre ces deux organes puisqu'on peut les construire ensemble. Il devient évident qu'une partie du nœud d'organisation se simplifie de fait, dans la matérialisation de cette notion, dans le Vecteur d'Information de l'Insertion Fractionnée, où toute la partie communication système-processeur devient directe. Mais il demeure néanmoins l'essentiel : l'information qui concerne le compilateur successeur.

En résumé, si l'on peut dire qu'on se place dans le domaine des concepts, on dispose d'une organisation globale implicite dans la mesure où elle est contenue dans la sémantique des jeux de langages qu'on utilise, avec condensée dans la partie système, une sémantique de répartition implicite qui complète la répartition explicite que l'utilisateur puise dans

ses langages pour pouvoir changer de plans de travail. Là, nous sommes hors machine dans la conception abstraite des programmes, je dirai dans l'espace abstrait de la programmation.

Quand on est face à la machine et que des programmes se déroulent je dirai qu'on est dans l'espace concret du déroulement. Des codes sont déroulés qui permettent de construire d'autres codes à partir d'un matériel symbolique. Il y a transformation d'une image symbolique en une image concrète, codée, déroulable. On se trouve dans l'espace concret de la programmation.

Je considère alors l'opération de déroulement elle-même, elle fait intervenir l'algorithme ultime, matérialisé, lui par du câblé. C'est l'espace concret du déroulement.

L'espace concret du déroulement est relié à l'espace concret de la programmation par deux sortes de liaisons : le code qui définit un plan d'organisation, et par le VIIF qui définit le nœud d'organisation.

Par contre, l'espace abstrait de programmation est relié à l'espace concret de programmation uniquement par le code, image réelle de la phrase abstraite.

A titre de conclusion je dirai simplement que le remplacement du concept d'organisation par le concept de complexité pourrait éventuellement permettre de se faire une idée de la façon dont on peut condenser des niveaux relatifs de complexité, à l'intérieur de systèmes éminemment complexes par le moyen de l'introduction de notions convenables dans l'organisation.

Note sur l'autojectivité et l'organisation

J'ai affirmé plus haut que l'autojectivité permet de réduire au minimum de complexité les rapports qui existent entre système et compilateurs. On peut s'attarder un peu sur cette réalité. Quel que soit l'algorithme que l'on considère, par exemple un compilateur, il existe toujours dans son organisation deux domaines distincts de complexité. D'abord on a toujours affaire à de la complexité inhérente au problème traité. Là déjà, la complexité se retrouve fonction de la qualité de l'analyse du problème théorique. Mais aussi une complexité apparaît quant au choix de la structure de l'algorithme, qui n'est fonction, elle, que de la nature de l'algorithme.

C'est ainsi qu'on peut définir deux grandes classes de structures algorithmiques : les algorithmes fibrés et les algorithmes en réseau.

La technique de l'algorithme en réseau a été la première utilisée pour la construction des compilateurs. Elle consiste simplement à noter un état de travail par une localisation dans l'algorithme. Par exemple un analyseur syntaxique qui fonctionne ainsi s'attend à rencontrer en un point de l'analyse tel jeu de lettres. S'il s'agit d'une lettre différente : erreur, sinon l'une des bonnes lettres renvoit dans une branche où l'on s'attend à un nouveau jeu de lettres etc. C'est donc parvenu en un point précis de l'algorithme qu'on peut savoir où on en est du traitement.

Un algorithme fibré ne traite qu'un seul état à la fois, mais il note dans une configuration les caractéristiques de cet état. C'est ainsi qu'un analyseur syntaxique qui rencontre une lettre consulte la valeur de l'état précédent pour l'admettre, et si elle est admise modifie l'état.

On peut bien entendu envisager tous les mélanges de ces deux méthodes.

On conçoit alors aisément qu'un algorithme fibré soit plus commode à rendre autojectif, dans la mesure où précisément il est déjà entièrement structuré en chemins parallèles.

Je dirai alors que tout algorithme si compliqué qu'il puisse être est doté de deux niveaux de complexité : la complexité intrinsèque, et la complexité de structure.

En quelque sorte, pour ce qui est de la compilation, mais c'est également vrai pour tout algorithme, la complexité intrinsèque reflète simplement la sémantique du langage traité, alors que la complexité de structure est l'image de l'adaptation au milieu dans lequel baigne le compilateur.

La démonstration de la propriété est simple : la structure de l'algorithme autojectif fait que celui-ci se déroule exclusivement par chemins élémentaires finis-bornés. Pour ce faire l'algorithme note dans sa configuration toute l'information dont il a besoin pour assurer la continuité de son travail d'un chemin sur l'autre. Cela lui laisse la charge entière de sa sémantique intrinsèque. En conséquence les relations qu'il aura obligatoirement avec le système seront réduites au maximum dans la mesure où ce dernier n'a plus qu'à se préoccuper de la gestion du fini-illimité : insérer autant de fois que nécessaire un chemin du compilateur, prendre en charge les demandes d'émission de messages ou d'échanges, éventuellement accorder des rallonges de place.

Un système qui utilise les interruptions pour gérer ses compilateurs est obligé de mettre en réserve l'ouvrage du compilateur interrompu pour pouvoir le lui restituer avant de l'insérer de nouveau. Et cela n'est pas forcément chose simple si, en toute généralité, le compilateur est construit après que le système ait été mis en place.

Notes

[1] Je rappelle brièvement la définition de l'autojectivité. Est autojectif un compilateur qui possède un point d'entrée unique et un point de sortie également unique, et tel que tout chemin qui joint le point d'entrée au point de sortie est fini borné dans le temps et dans l'espace. Cela signifie que le déroulement de chaque chemin possible demande un temps borné et calculable à l'avance, et que les données traitées lors de chacune de ces circonstances occupent un volume de mémoire borné et aussi calculable à l'avance. Un compilateur compact est un exemple de compilateur autojectif.

[2] Je rappelle brièvement la définition de la boucle à ouverture : quand je décide de munir une boucle d'une ouverture, je précise le nombre de tours qui sera déroulé au maximum sur un chemin. Ainsi, quand on pénètre dans une telle boucle on ne déroule que le nombre de tours précisé, et il y a retour dans le système au chemin suivant, on pénètre directement dans la boucle pour dérouler si possible seulement le nombre de tours précisé, et cela jusqu'à épuisement de l'argument.

Bibliographie

Le Bulletin d'Informatique Approfondie d'Applications, Editions du Litam, ISSN 0291-5413.