Les langages de spécification

M. DEMUYNCK* et B. MEYER**

Ceux à qui le Roi m'avoit confié remarquant combien j'étois mal habillé, donnerent ordre à un Tailleur de venir le lendemain, & de me prendre mesure pour un habillement complet. Cet Ouvrier le fit, mais d'une manière toute difererente de celle qui est en vogue en Europe. *Il prit d'abord ma hauteur à l'aide d'un quart de Cercle, & puis par le moyen d'une Regle & d'un Compas, il decrivit sur le papier toutes les dimensions de mon corps, & six jours après il m'apporta mes habits parfaitement mal faits, parce qu'il s'étoit mepris dans une Figure : Mais ce qui me consola, c'est que je remarquai que ces sortes d'accidens étoient fort ordinaires, & qu'on ne s'en mettoit guères en peine.*

[Swift] *Voyages du Capitaine Lemuel Gulliver en Divers Pays Eloignez, tome second, Première Partie, 1727.*

* Ingénieur-Chercheur au Centre de Documentation.

** Ingénieur-Chercheur au Département Méthodes et Moyens de l'Informatique.

E.D.F. – BULLETIN DE LA DIRECTION DES ETUDES ET RECHERCHES
SERIE C – MATHEMATIQUES, INFORMATIQUE n° 1, 1979, pp. 39-60.

TABLE DES MATIERES

I. INTRODUCTION

Les dernières années ont vu une prise de conscience croissante des résultats techniques et économiques en jeu dans ce qu'on a appelé la "crise du logiciel". Le seul chiffre de 250 milliards de francs – coût estimé de l'informatique dans le monde en 1975 –, et le fait que ces coûts sont à 75 %, et pour une proportion qui ne cesse de croître, des coûts de logiciel, montre qu'il est crucial d'améliorer les méthodes d'étude et de maintenance des systèmes informatiques [12]. Dans ce but, de nombreuses idées ont été proposées, qui ont nom "programmation structurée", "programmation modulaire", "programmation systématique", "analyse descendante", "analyse composite", etc. ; leurs partisans insistent sur la nécessité d'une démarche logique et rigoureuse, sur l'importance de la clarté et de la documentation des programmes, sur l'emploi de structures de contrôle et de données bien définies.

Bien que ces idées soient fructueuses et fondamentales, l'expérience montre qu'elles ne résolvent qu'une partie du problème. Lorsqu'en effet on en arrive à l'étape de programmation, tout est déjà joué pour une large part : les véritables difficultés se placent avant, au moment où l'on cherche à comprendre et décomposer le problème. L'importance de cette phase est particulièrement évidente dans un domaine comme l'informatique de gestion, où la plupart des tâches à effectuer sont conceptuellement simples ; cependant, même si les éléments du système, pris individuellement, semblent "faciles", leur nombre même et leur imbrication rendent extrêmement ardue la compréhension de l'ensemble.

Une des conséquences de cette situation, qui se produit dans tous les domaines d'application, est qu'il manque la plupart du temps un bon *Cahier des charges.* Le cahier des charges est trop souvent énorme, complexe, difficile à comprendre et difficile à contrôler.

Vérifier qu'il est cohérent et complet est une tâche impossible dans ces conditions. Le résultat le plus clair est l'apparition au stade de la programmation de difficultés et de choix qui auraient dû être réglés au cours de l'"analyse" et qui peuvent perturber gravement, voire mettre en péril, la réalisation du projet de programmatin.

Le concept de *langage de spécification* correspond à cette recherche d'une méthode pour *poser* les problèmes avant de commencer à les *résoudre.* Il s'agit de fournir un cadre précis pour constituer sous une forme fiable et rigoureuse, pouvant éventuellement être traitée automatiquement, l'exposé d'un problème que l'on envisage de résoudre sur ordinateur.

II. LES LANGAGES DE SPECIFICATION : UN ENSEMBLE DE CRITERES

II.1. Introduction

Depuis quelques années, plusieurs essais ont été faits pour proposer des langages de ce type et les travaux continuent dans ce domaine.

Les paragraphes suivants de cet article seront consacrés à discuter en quoi le concept de langage de spécification diffère du concept de langage de programmation (section III) ; à passer en revue quelques-uns des formalismes et systèmes proposés (section IV) ; à en décrire un autre, le langage Z de Jean-Raymond Abrial, que nous considérons comme l'approche la plus prometteuse (section V) ; et, en conclusion, à discuter la portée et les implications de cette approche, ses rapports avec les autres systèmes décrits et le bénéfice qu'elle peut en tirer.

II.2. Critères d'évaluation des langages de spécification

L'étude des différents formalismes sera fondée sur les critères suivants, qui nous paraissent les plus utiles pour notre approche comparative des langages de spécification.

II.2.1. Aspect statique/dynamique

Le langage décrit-il seulement des *problèmes* (aspect statique) ou est-il orienté vers l'indication de *processus* (aspect dynamique) ?

II.2.2. Niveau d'abstraction

Quel est le niveau d'abstraction du langage ? Permet-il à l'utilisateur d'oublier les détails des sous-sytèmes ? Jusqu'à quel point les spécifications prescrivent-elles une méthode d'implémentation particulière ? Est-ce qu'elles permettent au contraire de faire abstraction des techniques qui pourront être ultérieurement utilisées ?

II.2.3. Généralité

Le langage est-il conçu pour une classe particulière de machines, de systèmes, de langages de programmation ou d'applications ?

II.2.4. Modularisation

Le formalisme de spécification aide-t-il dans la décomposition d'un système en sous-unités, ou bien suppose-t-il que cette décomposition a déjà été faite ?

II.2.5. Méthodologie

Le langage inclut-il des règles méthodologiques qui aideront à produire des spécifications amenant des systèmes fiables, ou bien de telles règles doivent-elles être ajoutées comme des restrictions à l'utilisation de certaines facilités ?

II.2.6. Dispositifs automatiques

Le langage est-il prévu pour des spécifications "sur papier", ou bien est-il susceptible de donner lieu à un traitement automatique, en particulier à des fins de documentation ?

II.2.7. Base théorique et rigueur

Le langage a-t-il une base théorique saine ? Est-il défini seulement par des documents écrits en langue naturelle ? Existe-t-il une définition précise, éventuellement axiomatique ?

II.2.8. Cadre linguistique

Dans quelle sorte de formalisme sont exprimées les spécifications (langue naturelle, diagrammes, automates, tables, logique mathématique, théorie des ensembles. . .) ?

II.2.9. Description de la syntaxe/ de la sémantique

Certains formalismes ne permettent de décrire que la *syntaxe* d'un système, c'est-à-dire sa structure et les relations entre les éléments. Nous soutiendrons que la *sémantique,* à savoir le rôle et la signification de ces éléments, devrait aussi être incluse dans les spécifications, exprimée dans le formalisme lui-même.

II.2.10. Support graphique

Existe-t-il une aide visuelle (automatique ou manuelle) ? Nous discuterons le pour et le contre des formalismes graphiques au paragraphe VI.2.

II.2.11. Utilisateurs

A quel type d'utilisateurs est destiné le langage (non-informaticiens, analystes, programmeurs) ? La spécification d'un système sera-t-elle compréhensible par les personnes pour qui le système est développé ? Qui écrira et qui utilisera les spécifications ?

II.2.12. Apprentissage et facilité d'emploi

Le langage et la méthodologie associée sont-ils difficiles à apprendre/à enseigner ? L'utilisation du langage entraîne-t-elle un surcroît de travail fastidieux ?

II.3. Portée de l'étude

Plusieurs formalismes possibles seront brièvement examinés en rapport avec les critères ci-dessus mentionnés : langages de programmation, méthodes d'analyse (Warnier, Jackson), tables de décision, diagrammes de transition, réseaux de Pétri, diagrammes HIPO, ISDOS, SREM-SREP, SADT, types abstraits de données, et Z. Les limites imposées à un article font que nous nous sommes limités à des généralités sur tous les systèmes étudiés (sauf le dernier, que nous exposerons avec plus de détails) ; nous espérons cependant avoir dégagé les principaux apports de chacun. (Nous devons cependant mentionner qu'il est particulièrement difficile d'obtenir des informations spécifiques sur quelques-uns des systèmes américains les plus ambitieux ; ceci semble tout à fait courant dans ce domaine de recherche, sans doute pour des raisons commerciales, plus que scientifiques).

III. POURQUOI LES LANGAGES DE PROGRAMMATION NE SONT-ILS PAS ADAPTES A LA SPECIFICATION ?

Avant de commencer la description des langages de spécification, il est utile de discuter en quoi ce concept diffère de celui des langages de programmation. Pourquoi ne pourrait-on spécifier un problème en COBOL, PL/1, ALGOL, PASCAL ? Il est en effet intéressant de noter que les langages de programmation satisfont à une partie des critères requis. Ces langages sont définis de façon remarquablement précise et non ambiguë (au moins si on les compare à tout système non formel) ; leur nature les rend susceptibles d'un traitement automatique, et ils produisent des aides à la documentation intéressantes (tables de références croisées, listes de tables de symboles, etc.). Quelques-uns ont acquis un haut degré de normalisation, qui les rend indépendants d'une machine particulière ou d'un système d'exploitation. Enfin, l'expérience a montré qu'ils étaient assez facilement compréhensibles et pouvaient être enseignés sur une large échelle.

Ce qui rend les langages de programmation impropres à la spécification, c'est à la fois un trop grand *niveau de détail* et leur caractère essentiellement *dynamique*.

Le premier de ces deux points se traduit par l'obligation pour l'utilisateur de préciser trop de choses trop vite. Dans l'ordre des structures de données, par exemple, une déclaration PL/1 de la forme

```
DCL 1 COMPTE (1000),
    2 CREDIT,
    2 DEBIT,
    2 SOLDE ;
```

implique qu'on a choisi une fois pour toutes un certain mode de représentation de la structure des données ou du ficher, alors qu'il aurait pu être plus sage de recalculer les soldes à partir des

crédits et des débits chaque fois que l'on en avait besoin, ou d'utiliser une approche intermédiaire ("mémo-fonctions").

Le second problème mentionné (l'aspect dynamique, ou *procédural,* des langages de programmation usuels) est dû au fait que les langages de programmation obligent à décrire *comment* sont calculés les résultats, alors que l'objet d'une spécification se limite à la question "*que* calcule-t-on ?". Un des exemples les plus simples est, dans le domaine des structures de contrôle, l'obligation faite au programmeur, par la plupart des langages, de programmer séquentiellement, c'est-à-dire de choisir un ordre d'exécution, même si celui-ci n'a aucune importance pour le problème.

Ces deux "défauts" des langages de programmation (excès de détail et aspect dynamique) conduisent à se tourner vers d'autres formalismes pour les problèmes de spécification. Il est toutefois intéressant de noter que l'évolution des langages de programmation tend à réduire ces deux défauts par des *possibilités d'abstraction.* L'*abstraction procédurale,* fournie en particulier par les appels de sous-programmes, permet de donner un nom à un sous-processus en négligeant temporairement son contenu :

CALL XX (A, B, . . .)

La méthodologie actuelle de la programmation a systématisé cette idée sous le nom de *conception descendante.* L'*abstraction de données,* offerte par les langages de programmation les plus récents, procède de la même approche pour l'étude des données, en particulier en liaison avec la notion de *type abstrait de données* (cf. IV.6 ci-dessous). Enfin, les difficultés provenant du caractère trop "dynamique" des langages de programmation traditionnels ont été étudiées depuis longtemps en liaison avec le développement des *langages non-procéduraux,* particulièrement LISP et ses dérivés (et dans une moindre mesure, APL). Il est bien connu, par exemple, que pour certains types de problèmes, les définitions récursives permettent une expression de programmes plus proche de la notation mathématique que les formulations itératives. La programmation "fonctionnelle" non-procédurale a été discutée récemment par Backus [4].

IV. QUELQUES FORMALISMES DE SPECIFICATION

IV.1. Méthodes d'analyse et de conception (Warnier, Jackson)

Des *méthodes d'analyse et de conception* largement utilisées comme celles de *Warnier* [18] et *Jackson* [7] pourraient sembler avoir leur place ici. Leur examen et celui d'autres méthodes similaires montrent rapidement toutefois qu'elles ne sont pas adaptées à la spécification ; elles aident à organiser et structurer données et programmes (ceux-ci étant déduits de celles-là dans les deux méthodes citées), mais ne fournissent que peu de recours quant à la manière de poser le problème. Une bonne méthode de conception n'est pas nécessairement une bonne méthode de spécification.

IV.2. Tables de décision, diagrammes de transition, réseaux de Pétri

Un certain nombre de formalismes couramment utilisés pour la spécification et la conception des programmes ont en commun deux caractéristiques importantes : ils sont fondés sur une description par états de transitions et ils se prêtent aisément à une représentation graphique. Les tables de décision, les diagrammes de transition et les réseaux de Pétri appartiennent à cette classe.

Le principe des *tables de décision* [9] est de décrire un processus par les transitions à effectuer en fonction de la situation actuelle et d'une certaine combinaison de conditions externes (fig. 1).

Conditions					
	retrait demandé ⩽ solde	oui	oui	non	non
	autorisation de dépassement	oui	non	non	oui
Actions					
	autoriser le retrait	oui	oui	non	oui
	envoyer un avertissement	non	non	oui	oui

Figure 1. – Table de décision

Les tables de décision ont plusieurs avantages. Leur principe est simple et aisé à enseigner ; il n'y a pas d'ambiguités ; elles obligent le concepteur à des vérifications, puisque toutes les combinaisons possibles d'entrée doivent être examinées (malheureusement, cela semble être une pratique courante que de laisser beaucoup de positions de la table en blanc, c'est-à-dire non spécifiées) ; elles se prêtent à la vérification automatique ; elles peuvent être traduites en programmes ("conversion des tables de décision") ; comme nous le verrons, il existe des moyens naturels de les représenter graphiquement. Du point de vue de la spécification, cependant, elles sont beaucoup trop procédurales et algorithmiques, prescrivant un enchaînement rigoureux d'actions. Par ailleurs, l'emploi des tables de décision amène dans tout problème un peu compliqué à considérer un nombre considérable de combinaisons, et il devient difficile de conserver le contrôle du processus conceptuel, même par l'emploi de sous-tables.

Une idée voisine est celle des *diagrammes de transition*, que l'on rencontre couramment dans le domaine de la compilation. Initialement appliquée à la description de la structure lexicale des langages de programmation, c'est-à-dire des langages réguliers (ou automates finis), elle a été étendue par Wirth, dans la description de Pascal [8], aux langages sans contexte (fig. 2).

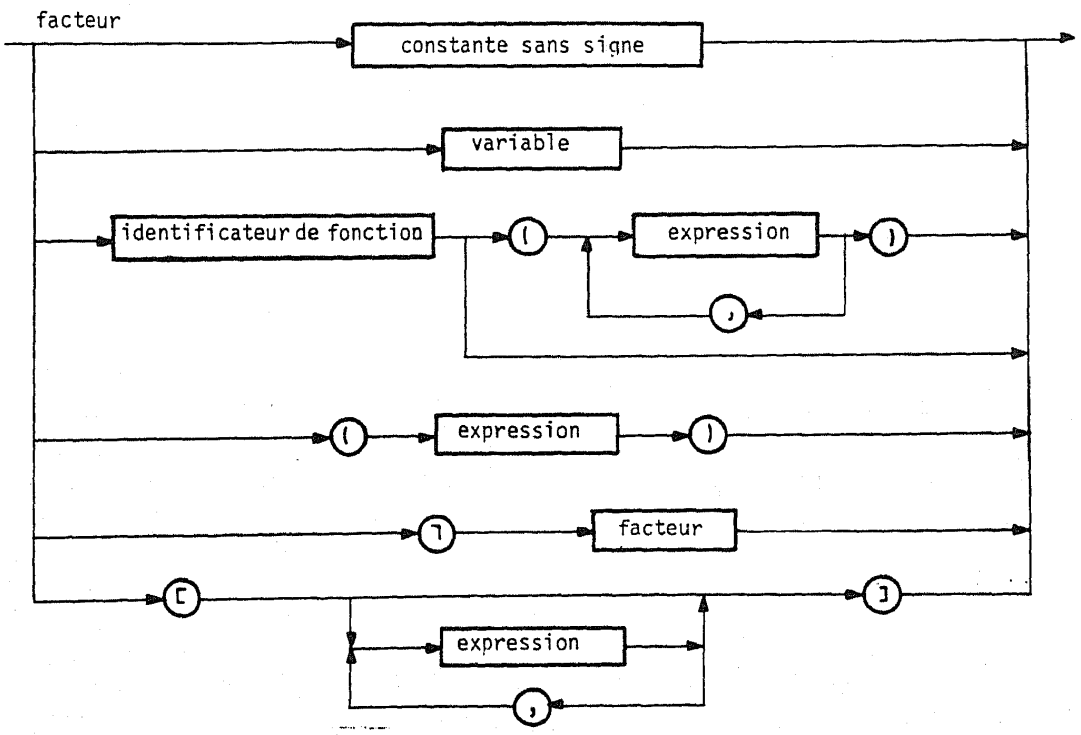

Figure 2. – Diagramme syntaxique (tiré de la définition de Pascal) [8]

Ce type de formalisme se prête bien à la représentation de tout processus de type "états-événements" (fig. 3).

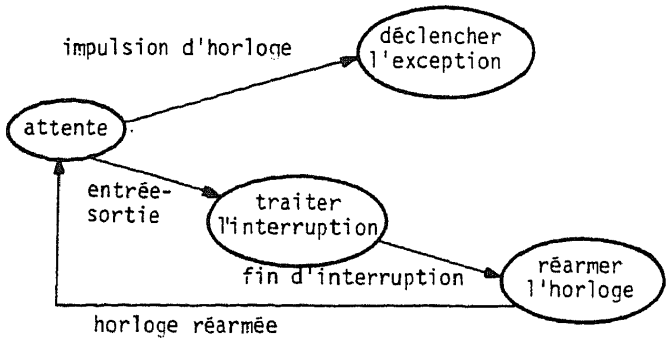

Figure 3

Les diagrammes de transition correspondent à une technique classique de la programmation, celle de la "commande par table" [12], permettant de reporter une partie de la commande du programme dans ses données, les avantages essentiels étant une grande souplesse, la possibilité de changer facilement certains paramètres du processus, et d'obtenir des systèmes évolutifs.

Pour le reste, ils présentent les mêmes avantages et les mêmes défauts que les tables de décision, auxquelles ils fournissent une représentation graphique commode.

Les concepteurs de systèmes parallèles et en temps réel préfèrent généralement utiliser un formalisme graphique particulier, les *réseaux de Pétri* [14]. Un réseau de Pétri (fig. 4) permet de décrire le fonctionnement d'un système, représenté par des *places* qui peuvent abriter un certain nombre de *jetons* associés à des processus ou des événements, et des *transitions* reliant différentes places. Une transition ne peut se "déclencher" que si toutes ses places en entrée sont remplies par un jeton ; son déclenchement envoie alors un jeton à chacune des places en sortie. Si plusieurs transitions peuvent se déclencher à un certain moment, on ne peut prévoir laquelle sera exécutée ; cet aspect non-déterministe est fondamental dans la modélisation des processus parallèles.

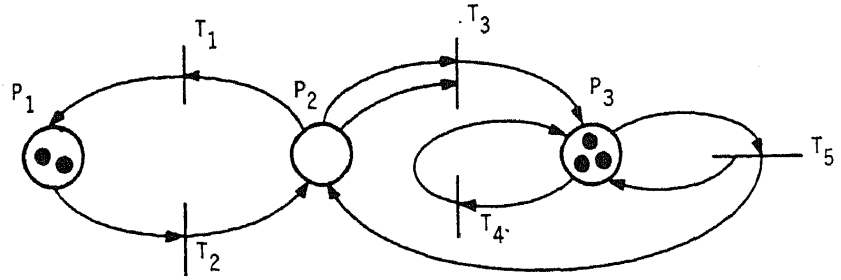

Figure 4. – Réseau de Pétri

Comme les tables de décision, les réseaux de Pétri ont l'avantage de pouvoir être traités et vérifiés automatiquement. Comme elles, cependant, ils conduisent aisément à des descriptions complexes et peu structurées. De même que les organigrammes en programmation, il s'agit d'outils très riches, dans lequels les mécanismes de structuration ne sont pas incorporés, mais doivent être ajoutés *a posteriori*.

IV.3. HIPO

HIPO [16] est une méthode de conception proposée par IBM et fondée sur l'emploi de diagrammes *Hi*érarchisés décrivant, pour chaque élément d'un système, ses entrées (*I*nput), le traitement à effectuer (*P*rocess) et ses sorties (*O*utput) (fig. 5).

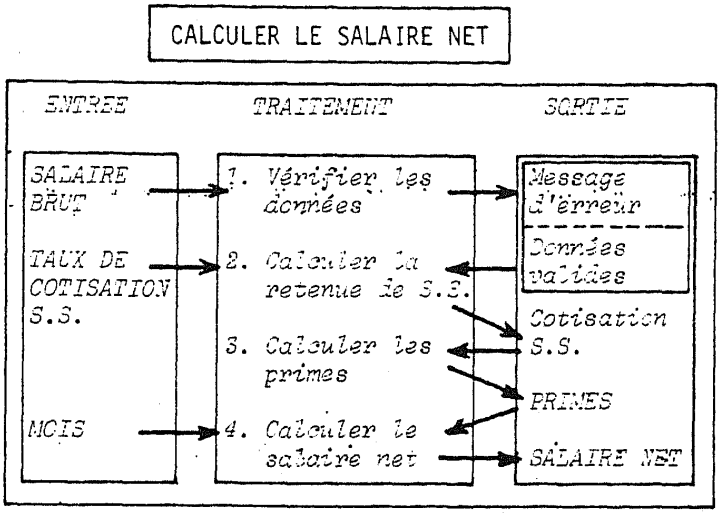

Figure 5. – Diagramme HIPO (schéma de principe)

HIPO permet de décrire un système de façon détaillée et complète. Du point de vue qui nous intéresse, cependant, on notera qu'il s'agit d'une description de caractère trop "procédural" pour être tout à fait satisfaite en tant que spécification. Par ailleurs, la notion de "traitement" demande à être précisée ; les diagrammes HIPO désignent les éléments de traitement par des phrases en langue naturelle ("calculer la somme à payer", etc.), avec leur inévitable manque de précision et leur risque d'ambiguité. D'un point de vue pratique, remplir les diagrammes HIPO semble assez fastidieux. Enfin, la méthode HIPO suppose que l'on a auparavant séparé le traitement en modules, bien que les critères pour une telle décomposition soulèvent des problèmes plus difficiles que ceux rencontrés lors de la description de modules simples ; la meilleure décomposition possible n'est de toute façon probablement pas fondée uniquement sur le traitement, mais aussi sur les structures de données (cf. "Types abstraits de données", section IV.6 et les références [13] et [12] ; le même problème se pose avec la méthode de conception IPT d'IBM). Ainsi HIPO peut être utilisé conjointement avec une autre technique de "modularisation".

IV.4. ISDOS, SREM-SREP

Plusieurs méthodes de spécification et de conception des systèmes informatiques ont été proposées aux Etats-Unis ces dernières années, parmi lesquelles deux paraissent relativement voisines : ISDOS, développée à l'Université du Michigan [17], et SREM-SREP développée par TRW [3]. Elles sont fondées sur l'utilisation d'organigrammes de fonction ("R-nets" dans [3]) assez voisins des formalismes de la section IV.2 (diagrammes de transition, réseaux de Pétri), mais visent à limiter la complexité par la restriction à un petit nombre de modes de combinaison imités des figures de base de la programmation structurée. L'un au moins de ces systèmes (SREM-SREP) offre une double forme d'expression : graphique et textuelle. L'un des objectifs essentiels de ces systèmes est de permettre un traitement automatique des spécifications, et en particulier de multiples vérifications de validité (vérifier que chaque objet est défini avant d'être utilisé, etc.).

ISDOS semble être fondé sur les mêmes principes qu'HIPO, mais permet en outre de traiter le problème de la décomposition en modules par la définition d'"interfaces", et de prendre en compte des informations sur l'environnement technique et humain. La "sémantique" du système est décrite par des commentaires en langage naturel.

Nous ne possédons pas suffisamment d'informations sur ces systèmes pour les analyser en profondeur. Il semble toutefois que, de notre point de vue de la spécification, ils soient trop "orientés traitement" et pas assez "statiques" pour être parfaitement adaptés à des spécifications véritables. Ils manquent en outre de rigueur dans leur définition (ISDOS en particulier inclut des concepts redondants d'"interface", "entrées et sorties" etc.), et semblent se limiter à la description de la "syntaxe" d'un système.

IV.5. SADT

SADT, développé par Softech [11] est en fait une "*T*echnique *S*tructurée" couvrant l'*A*nalyse et la Conception (*D*esign) hiérarchique, et incluant un langage de spécification. Ce langage est fondé sur des schémas duaux : les diagrammes de traitement appelés "actigrammes" et les diagrammes de données appelés "datagrammes". Chaque diagramme est constitué d'un nombre restreint (3 à 6) de boîtes reliées par des flèches. Pour chaque boîte, les flèches en entrée sont de trois types : *support* (machine réelle ou virtuelle), *entrée* et *contrôle* (la distinction entre ces deux derniers types ne semble pas très claire) ; les flèches en sortie représentent les *sorties* (fig. 6).

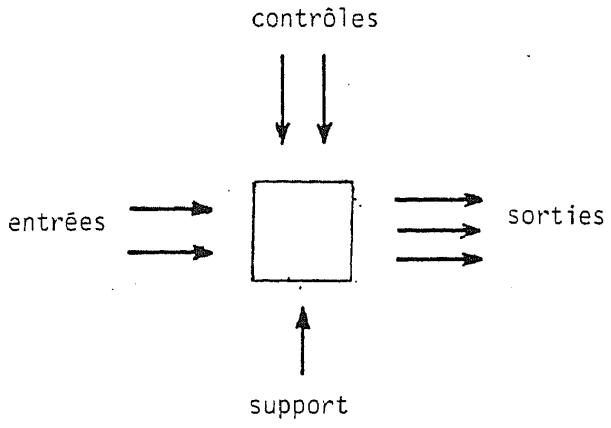

Figure 6. – Elément d'un schéma SADT

Le langage est orienté vers une conception hiérarchique modulaire et cherche à aider le concepteur dans le processus de décomposition. Ce système permet des sorties graphiques et, semble-t-il, des vérifications automatiques.

Les deux critiques principales qui peuvent être faites sont, d'abord, que l'utilisation des diagrammes SADT demande, comme dans le cas de HIPO, un codage assez lourd et fastidieux ; par ailleurs, et ceci est plus important, que SADT, comme les diagrammes de transition et les réseaux de Pétri, permet essentiellement la description de ce que nous avons appelé la syntaxe d'une spécification (c'est-à-dire l'organisation structurelle des modules et leurs relations), à l'exclusion de la sémantique, c'est-à-dire qu'on ne trouve pas d'informations sur la manière dont la sortie d'un élément peut être déduite des entrées et des contrôles. Bien sûr, cette information peut être, et parfois est, exprimée au moyen de "phrases" en langue naturelle accompagnant les diagrammes ; toutefois, le procédé consistant à surajouter ainsi la sémantique est peu pratique, et risque d'entraîner des erreurs du fait qu'il n'est pas cohérent avec le formalisme de base.

SADT pourrait sans doute servir de support graphique utile à la représentation des formalismes que nous allons décrire maintenant.

IV.6. Les types abstraits

La notion de *Type Abstrait de Données* [10] [11] [12] [6] permet de caractériser les objets manipulés par les programmes de façon indépendante des problèmes de représentation physique. Une structure de données sera ainsi définie par la liste des opérations qui permettent d'y accéder et de la manipuler, ainsi que par les propriétés formelles, externes, de ces opérations (fig. 7).

type : *COMPTE*

fonctions :

créer : → *COMPTE*
ajouter : *COMPTE* x *ARGENT* → *COMPTE*
retirer : *COMPTE* x *ARGENT* → *COMPTE*
solde : *COMPTE* → *ARGENT*

propriétés

pour tout compte c et toute somme a :

solde (créer) = 0
solde (ajouter (c, a)) = solde (c) + a
solde (retirer (c, a)) = solde (c) − a

Figure 7. – Une définition "fonctionnelle" d'un type abstrait

Une conséquence intéressante de cette approche est qu'elle conduit naturellement à une décomposition des systèmes fondée sur les structures de données plutôt que sur les éléments de traitement [13].

On obtient, par cette méthode, des spécifications précises, qui aident à décomposer et à concevoir des programmes "bien structurés". La plus grande partie de la recherche actuelle sur les langages de programmation est consacrée à la conception de langages qui mettent en application (en développant les concepts de SIMULA 67) l'idée de type abstrait de données. De tels langages devraient faciliter l'implantation de systèmes spécifiés de cette manière.

Nous ne discuterons pas plus avant ce type de spécifications, car les types abstraits peuvent être décrits dans le formalisme qui va être décrit ci-dessous, le langage Z.

V. LE LANGAGE Z

V.1. Introduction

Le langage Z, créé par J.R. Abrial [1] [2] est un formalisme pour la spécification, la conception et l'écriture des programmes, visant à permettre à l'utilisateur de conserver à tous les niveaux de l'analyse la maîtrise complète des processus en jeu. Z est fondé sur quelques idées importantes que nous allons maintenant décrire.

La première idée de base est qu'un programme doit être développé par *étapes successives*, permettant petit à petit de se rapprocher d'un système donné et d'un langage de programmation, en partant des spécifications du problème. Z permet de décrire le problème de façon totalement indépendante d'une implémentation éventuelle ; des transformations successives introduisent des concepts algorithmiques dans la spécification et permettent d'obtenir de façon sys-

tématique des programmes écrits dans un langage donné. La seconde idée fondamentale est l'importance attachée à la *spécification statique* d'un problème. L'expérience montre, nous l'avons vu, que l'une des principales difficultés d'un projet informatique est de produire une description statique, rigoureuse et complète, du problème à résoudre, matérialisée par un cahier des charges qui ne soit pas obscurci par un aspect trop analytique, par une trop grande abondance d'informations, par des incohérences fréquentes et par un mélange confus de propos provenant de critères aussi différents que la structuration des données, les contraintes de temps, et les méthodes d'implémentation. Z permet de telles descriptions et permet de les considérer comme des programmes bien que ne contenant aucun élément dynamique, c'est-à-dire rien qui spécifie plus qu'il n'est nécessaire le déroulement dans le temps du traitement informatique.

Une troisième idée importante est que, parmi tous les formalismes possibles pour représenter les objets informatiques et leurs relations, il en existe un plus simple, plus clair et plus général que les autres, parce que résultant de plusieurs siècles d'évolution et de mise à l'épreuve : c'est le formalisme mathématique. Z est donc fondé sur les concepts et les notations de la *théorie des ensembles.* Un certain nombre d'autres sources ont influencé la conception de Z, permettant de traiter les problèmes spécifiques rencontrés en programmation :

– les notations des *langages de programmation* de la famille ALGOL, qui caractérisent la forme externe des spécifications Z ;
– les *modèles relationnels* de bases de données, introduits par Codd ;
– le *lambda-calcul,* déjà utilisé dans la conception de LISP et BCPL ;
– les *types abstraits de données,* introduits par Liskov et Zilles ;
– et, bien sûr, la *programmation structurée.*

V.2. Structure d'une spécification en Z

A partir d'un cahier des charges, le "spécificateur" écrira des *clauses* Z, représentant chacune une formalisation partielle d'une petite partie de ce document.

Il y a 3 sortes de clauses :

V.2.1. ***Les clauses de TYPE*** qui introduisent les différentes sortes d'objets qui apparaissent dans le système. Par exemple :

```
type
    LIVRE ;
    LECTEUR some = (moi) ;
    CATEGORIE_DE_LIVRES = {nouvelle, monographie, roman, bande_dessinée} ;
    ordered MOIS = {janvier, février, . . . , décembre} ;
```

V.2.2. ***Les clauses de RELATION*** qui définissent les relations (ou fonctions) existant entre les types définis précédemment. Par exemple :

```
relation
    type_de_livre : LIVRE → CATEGORIE_DE_LIVRES ;
    propriétaire_du_livre : LIVRE → LECTEUR ;
```

V.2.3. ***Les clauses d'ASSERTION*** qui expriment les lois logiques du système. Par exemple :

```
assertion
    forall b in LIVRE then
    |       if propriétaire_du_livre(b) = moi then
    |       |       type_de_livre(b) ≠ bande dessinée
    |       end
end;
```

Du point de vue de la théorie des ensembles, on peut considérer grossièrement les "types" comme des ensembles d'objets ; les "relations", comme des fonctions (mono ou multivaluées) entre ces ensembles ou des combinaisons d'ensembles ; et les "assertions" comme des axiomes dans une théorie logique. Les clauses de type et de relation décrivent la structure, ou syntaxe, d'un système ; les assertions, sa sémantique.

Les seules règles sont qu'une clause doit être compréhensible à partir des précédentes et que deux clauses ne doivent pas se contredire.

La méthode à utiliser pour écrire une spécification Z est itérative et de nature descendante. La première étape sera consacrée à une lecture soignée et répétitive du cahier des charges. La première lecture permettra d'obtenir les types de base du problème, quelques relations et peu d'assertions. La lecture suivante déterminera tous ou presque tous les types, et quelques nouvelles relations et assertions. Des lectures ultérieures permettront d'affiner les types d'objets, les relations et les assertions.

Une des caractéristiques les plus agréables de Z est que, grâce à la manière dont le langage a été conçu, les questions qui surgissent au cours des étapes sont les questions véritablement importantes pour une compréhension complète du problème. Autrement dit, on doit répondre à la bonne question au bon moment. Ces questions pourront être des types suivants :

– Existe-t-il une *relation d'ordre intrinsèque* entre les objets d'un type (par exemple, dans les mois d'une année, mais peut-être pas dans l'ensemble des transactions bancaires) ?

– Une certaine relation est-elle *fonctionnelle* (c'est-à-dire monovaluée) ?

Par exemple :

$$PARENTS : PERSONNE \rightarrow PERSONNE$$ ne l'est pas ;

$$époux : PERSONNE \rightarrow PERSONNE$$ l'est.

– Une certaine relation est-elle *totale* ou *partielle* (*âge* et *sexe* sont totales, *époux* ne l'est pas) ?

– L'inverse d'une relation a-t-il une signification pour le problème, comme dans :

$$appartient\_à : LIVRE \leftrightarrow BIBLIOTHEQUE : \textbf{set}\ CATALOGUE$$

(où la fonction inverse *CATALOGUE* est multivaluée).

– Quelles sont les propriétés ("assertions") d'une relation donnée ? Il est très instructif dans la pratique d'essayer de les écrire formellement et de découvrir des erreurs ou des incohérences dans le cahier des charges !

Quand les itérations sur le texte sont terminées, la spécification formelle est considérée comme une description complète et cohérente du système : ce sera la première *version* du système.

Toutefois, la spécification n'est pas terminée : des transformations et des améliorations vont suivre. Elle seront générales et systématiques ; la seule contrainte est que toutes les versions devront rester sémantiquement équivalentes. On trouvera notamment les transformations suivantes :

– Groupement de relations (représentant des fonctions totales) en une relation dont le domaine est le produit cartésien des domaines initiaux ;

– Elimination du non-déterminisme (un ensemble fini peut, par exemple, être "implémenté" comme une suite ordonnée finie) ;

– Elimination de la récursion ;

– Choix d'une méthode d'accès particulière pour les éléments d'un certain type (qui seront représentés par des fichiers ou des structures de données) ;

– etc.

De chaque transformation résulte une nouvelle *version* de la spécification formelle.

V.3. Description succincte du langage

Z comprend deux parties : le noyau, qui est théoriquement suffisant, et les extensions syntaxiques qui sont des constructions utiles en pratique, mais pouvant être exprimées en termes d'éléments du noyau.

V.3.1. Le noyau

V.3.1.1. Types

Les types en Z sont de 3 sortes :

– Certains types sont prédéfinis, à savoir ***BOOL, NUM*** et ***CHAR*** ;

– D'autres sont définis par leur occurence dans une clause de type ;

– D'autres encore peuvent être construits par composition à partir des précédents, à l'aide des opérateurs suivants :

- ***set*** (X) est le type dont les éléments sont les sous-ensembles de X ;
- ***tuple*** (X) est le type dont les éléments sont toutes les suites ordonnées d'objets du type X ;
- ***prod*** $(X_1, \ldots, X_n)$ est le produit cartésien de $X_1, \ldots, X_n$;
- ***function*** (X, Y) est le type dont les éléments sont toutes les fonctions monovaluées totales de X vers Y.

Tous les types sont disjoints.

V.3.1.2. Clauses

Les clauses de spécification sont soit des clauses de type, soit des clauses de relation, soit des clauses d'assertion. Deux autres sortes de clauses sont utiles en pratique : les clauses de définition et les clauses de partition.

a) *Les clauses de type* permettent la définition de nouveaux types, soit en extension, par exemple :

type
FEUX_DE_CIRCULATION = {*vert, orange, rouge*};

soit en ne donnant que le nom de quelques éléments, par exemple :

type
CLUB_DE_Z ***some*** = {*demuynck, meyer*} ;

soit en ne donnant que le nom du type si aucun élément n'est connu à l'avance :

type
ENREG_EN_ENTREE ;

b) *Les clauses de relation* définissent des relations entre types. Les fonctions monovaluées (totales sauf si ***partial*** est précisé) sont écrites en minuscules :

relation
impôt : ***prod*** *(CONTRIBUABLE, ANNEE)* → ***NUM*** ;

tandis que les fonctions multivaluées sont écrites en majuscules et précédées du mot ***set*** (ou ***ordered set*** si l'ordre a une signification pour le résultat) :

```
relation
    set FACTURES_DU_MOIS : CLIENT → FACTURE ;
    ordered set CANDIDATS_ACCEPTES : EPREUVE → CANDIDATS ;
```

Si la relation inverse a une importance pratique, elle apparaîtra, avec ses caractéristiques, à la droite de la relation :

```
relation
    numéro_de_voiture : VOITURE ↔ NUMERO : partial voiture_de_numéro
```

c) *Les clauses d'assertion* expriment les propriétés sémantiques des objets de spécifications. Elles correspondent en général à des prédicats. Les opérateurs comprennent ***forall*** :

```
assertion
    forall p in ENTREPRISE then
    |       age(p) ⩽ 65
    end ;
```

et ***if***, comme dans l'exemple suivant qui montre aussi l'utilisation de l'opérateur ***given*** :

```
assertion
    forall p in ENTREPRISE then
    |   given
    |   |       limite = if sexe(p) = masculin then
    |   |                |       65
    |   |                elsif sexe(p) = féminin then
    |   |                |       60
    |   |                end
    |   then
    |   |       age(p) ⩽ limite
    |   end
    end ;
```

D'autres opérateurs s'appliquent aux relations. ***lambda*** permet la définition de fonctions :

```
assertion
    valeur_absolue = lambda x in NUM ⇒
    |                if x ⩾ 0 then
    |                |   x
    |                elsif x < 0 then
    |                |   −x
    |                end
    end ;
```

closure donne la fermeture transitive d'une relation. Cet opérateur est particulièrement important car il permet d'introduire le concept de boucle sans perdre le caractère statique de Z.

d) *Les clauses de définition* sont utilisées comme facilité pour nommer des objets, c'est-à-dire comme macros. Elles sont souvent employées en liaison avec l'opérateur ***subset*** :

```
définition
    REGULIERE = subset m in MATRICE where
    |           déterminant (m) > 0
    end ;
```

REGULIERE n'est pas un type, mais un sous-ensemble du type *MATRICE* ; une convention différente aurait contredit le principe de disjonction des types.

e) *Les clauses de partition* permettent d'exprimer un type comme une union de sous-ensembles disjoints :

```
partition
    EMPLOYES = (COLS_BLANCS, COLS_BLEUS) ;
    EMPLOYES = (EMPLOYES_MASCULINS, EMPLOYES_FEMININS) ;
```

V.3.2. Extensions syntaxiques

Les extensions syntaxiques servent à définir des constructions utiles en pratique, mais qui ne sont pas théoriquement indépendantes de celles du noyau. Par exemple, l'opérateur ***exist*** :

```
exist p in ENTREPRISE where
    |       rang(p) = chef
end ;
```

est défini à partir de l'opérateur ***forall.*** De même, l'opérateur "union d'ensembles" est défini à partir de l'opérateur "intersection d'ensembles" (du fait des contraintes de validité des types, l'union peut seulement être appliquée à des sous-ensembles d'un même type)

V.4. Un exemple

A titre d'exemple, nous incluons ici une courte spécification en Z. Les commentaires sont parenthésés par /* et */.

```
spécification paie ;
/* Le but de l'exemple est d'obtenir une liste des salaires dans une entreprise, un mois donné,
triés par nom d'employé */
    type
        ENTREPRISE, MOIS, EMPLOYE ;
    relation
        salaire : prod (EMPLOYE, MOIS) → NUM ;
        nom : EMPLOYE → tuple (CHAR) ;
        appartenance : EMPLOYE ↔ ENTREPRISE : set PERSONNEL ;
/* On introduit ici une description de l'enregistrement de sortie et des moyens de l'obtenir */
    type
        ENREG ;
    relation
        nom_enr : ENREG → tuple (CHAR) ;
        salaire_enr : ENREG → NUM ;
        liste_salaire : prod (ENTREPRISE, MOIS) → tuple (ENREG) ;
/* pour obtenir une "liste_salaire" triée, définissons la notion de "trié" : */
    relation
        trié : tuple (ENREG) → BOOL ;
/* pour ceci, définissons tout d'abord l' "ordre alphabétique" : */
```

```
definition
    CHAINE = tuple (CHAR) ;
relation
    ordalph : prod (CHAINE, CHAINE) → BOOL ;
assertion /* "ordalph (t1, t2)" signifie "t1 est inférieur ou égal à t2 dans l'ordre alphabétique" */
    ordalph =
        lambda s1, s2 in prod (CHAINE, CHAINE) ⇒
        |  s1 = null or
        |  (s1 ≠ null and s2 ≠ null and
        |     (first (s1) < first (s2) or
        |     (first (s1) ≠ first (s2) and ordalph (tail (s1), tail (s2)))))
        end ;
/* Une définition non récursive aurait aussi été possible. La définition de "trié" est alors : */
    assertion
        trié =
            lambda r in ENREG ⇒
            |       forall v in r then
            |       |     ordalph (nom_enr (v), nom_enr (next (v))
            |       end
            end ;
/* next (v) désigne l'élément qui suit v dans le tuple. Définisssons maintenant la sortie désirée : */
    assertion
        forall e in ENTREPRISE then
        |   given
        |   |   ℓ = liste_salaire (e)
        |   then
        |   |   trié (ℓ) and
        |   |   forall p in PERSONNEL (ℓ) then
        |   |   |   p in ℓ
        |   |   end
        |   end
        end ;
```

V.5. Portée, utilisation et problèmes de Z

Z a été utilisé jusqu'ici sur une échelle relativement modeste. Il a été appliqué à la spécification de systèmes informatiques de gestion ; il est employé comme outil dans un travail en cours dont le but est d'améliorer la compréhension et la mise en œuvre des algorithmes numériques, et a été appliqué à la spécification d'autres programmes, comme des traducteurs de macros. L'expérience a montré aussi que Z était applicable de manière tout à fait satisfaisante à la modélisation des systèmes parallèles, comme les processus "producteur-consommateur", les systèmes décrits par les résaux de Pétri, etc. Nous avons aussi trouvé Z intéressant comme outil théorique pour l'étude des processus de calcul.

Formalismes / Critères	Langages de programmation	Tables de décision	Diagrammes de transition	Réseaux de Pétri	HIPO	Warnier Jackson	SREM-SREP	ISDOS	SADT	Types abstraits de données	Z
Statique ou dynamique	Dynamique	Habituellement dynamique, mais peut-être statique	Dynamique	Dynamique	Dynamique	Dynamique	?	Plutôt dynamique	Statique	Statique	Statique
Niveau d'abstraction	Des facilités d'abstraction existent, mais tous les détails doivent finalement être donnés	Bas	Bas	Bas	Bas (dépend de l'utilisation)	Des facilités d'abstraction existent, mais tous les détails doivent finalement être donnés	Haut (?)	Haut	Haut	Haut	Haut
Généralité (c'est-à-dire portée des domaines d'application possibles)	Dépend du langage	Général	Général	Systèmes parallèles et temps-réel	Plutôt orienté vers les problèmes de gestion	Réputé pour être général mais très orienté vers les problèmes de gestion	Général plutôt orienté vers les problèmes temps-réel	Orienté vers les problèmes de gestion	Général ?	Général	Général
Aides dans le processus de décomposition ?	Oui, au moins dans les bons langages	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui
Existence d'une (bonne) méthodologie de spécification et de conception ?	Normalement non (des efforts récents vont dans cette direction)	Non	Non	Non	Mieux que rien	Oui	Oui	Oui	Oui	Oui	Oui
Base théorique saine et rigoureuse ?	En partie pour quelques langages	Oui	Peut être définie	Oui	Non	Non	?	Non	Non	Oui (algèbres initiales)	Oui
Cadre linguistique		Tables	Diagrammes (ou automates)	Diagrammes (ou automates)	Langage naturel	Arbres (plus quelques constructions des langages de programmation)	Diagrammes plus quelques constructions des langages de programmation	Peu clair	Diagrammes	Théorie des ensembles, calcul des prédicats	Théorie des ensembles calcul des prédicats, syntaxe type-ALGOL
Description de la "syntaxe" la "sémantique", ou les deux ?	Les deux	Les deux	Syntaxe	Syntaxe	Les deux	Les deux	Les deux (?)	Syntaxe	Syntaxe	Les deux	Les deux
Support graphique ?	Non	Non	Oui	Oui	Non	Arbres	Oui	Non (?)	Oui	Non	Non
Utilisateurs	Programmeurs	Utilisateurs, analystes, programmeurs	Utilisateurs, analystes, programmeurs	Analystes	Analystes	Analystes, programmeurs	?	Analystes	Analystes	Analystes, programmeurs	Utilisateurs, (?!) analystes, programmeurs.
Facilité d'enseignement	Facile	Facile	Facile	Modérément difficile	Facile	Plutôt facile	?	?	?	Difficile	Modérément difficile

Figure 8

Z semble applicable de façon fructueuse à tous les domaines d'application possibles en informatique. Toutefois, un plus large champ d'expérience nous manque manifestement pour justifier cette affirmation. Une telle expérience apporterait sans aucun doute des changements et des améliorations au langage.

Un des aspects de Z qui pourrait devenir fondamental mais demande beaucoup de travail est celui des transformations systématiques, qui permettent d'affiner une spécification en en préservant la sémantique et d'arriver petit à petit à un programme réaliste. Un catalogue d'environ 400 transformations a été écrit [2], qui se réfère à une ancienne version du langage ; toutefois, leur nombre même suggère qu'une meilleure approche reste à trouver – à moins que l'on considère que les transformations ne représentent que des équivalents bien exprimés des bons vieux "trucs" du programmeur, qui sont certainement aussi nombreux.

VI. RESUME ET CONCLUSIONS

VI.1. Résumé de l'étude comparative

Le tableau de la figure 8 montre le résultat obtenu en appliquant les critères définis dans la section II.2 aux formalismes étudiés. Le signe "—" signifie que le critère est sans signification pour le formalisme en question ; le signe "?" que nous manquons d'information sur ce formalisme pour ce critère.

L'étude montre de façon évidente que le problème des langages de spécification est loin d'être résolu. Nous pensons toutefois que quelques conditions nécessaires fondamentales ont été mises en évidence pour ces langages et nous allons les résumer.

VI.2. Conditions pour un bon langage de spécifications

Tout d'abord, pour être tout à fait digne du nom de "langage de spécification", un formalisme doit être statique et non-algorithmique, c'est-à-dire laisser de côté tous les aspects procéduraux.

Ensuite, il doit permettre d'exprimer à la fois la structure "syntaxique" d'un système et sa sémantique. Mais ce but ne doit pas être atteint aux dépens du précédent, c'est-à-dire que la description sémantique doit être non-procédurale. De tous les formalismes étudiés, les types abstraits de données et Z nous semblent les mieux à même de remplir ces objectifs.

Un autre problème est celui des supports graphiques, qui sont des éléments inclus à la base dans les diagrammes de transition, les réseaux de Pétri, SREM-SREP et SADT, mais non pour les tables de décision, les types abstraits de données ou Z. L'utilisation de tels supports met en avant un problème très général. Bien qu'ils soient très agréables au premier abord ("mieux vaut un bon dessin. . .") ils peuvent aussi être trompeurs, car la quantité d'information qu'on peut inclure dans un dessin est très limitée, même si c'est l'information la plus importante et si elle apparaît très clairement. Nous pensons par exemple que le problème principal de SADT est dû au fait que l'information sémantique ne peut faire partie des diagrammes eux-mêmes, mais doit être écrite en phrases du langage naturel, ce qui détruit l'intégrité conceptuelle du modèle.

D'un autre côté, les vertus pédagogiques des dessins sont telles qu'il serait dommage de ne pas essayer de les utiliser à des fins d'éclaircissement. Aussi, pouvons-nous hasarder la proposition suivante : Utiliser quelque chose comme SADT comme support graphique pour vulgariser et enseigner des spécifications écrites, transformées et contrôlées dans quelque chose comme Z.

Un autre problème est celui de la rigueur. Nous pensons qu'un langage de spécification doit avoir une base théorique bien définie et de taille restreinte – ce qui peut aussi impliquer que le langage sera trop mathématique, et difficile à comprendre pour des praticiens. Le problème débouche sur un autre, que nous n'avons pas étudié : qui écrira les spécifications et qui les utilisera ? Dans l'approche Z, comme dans la plupart des autres, sauf les plus simples (tables de décision, diagrammes de transition), la spécification est rédigée par un spécialiste, et peu d'utilisateurs finaux seront assez "sophistiqués" pour comprendre une spécification écrite dans ce langage. Ainsi, l'organisation devra comprendre un "médiateur" capable de retranscrire la spécification dans les propres termes de l'utilisateur final.

Parmi toutes les approches étudiées, Z est manifestement celle qui nous semble la plus prometteuse, et ce pour de nombreuses raisons : Z est applicable à tous les domaines de l'informatique ; il est statique et non-procédural, mais prépare cependant le terrain pour des transforma-

tions qui aident à parvenir aux programmes ; il a une base théorique saine ; il est complet, c'est-à-dire que les spécifications peuvent être entièrement exprimées à l'intérieur du cadre du langage ; et il combine la précision des mathématiques avec l'élégance d'expression des langages de programmation actuels.

Il est clair, cependant, que Z ne bénéficie pas d'une aussi large expérience que tous les autres systèmes étudiés. Il ne serait pas réaliste, en outre, d'ignorer les problèmes psychologiques et techniques qui peuvent se produire lors de l'introduction d'un tel formalisme. Nous pensons toutefois que si un langage de cette espèce se répand, la fiabilité et la souplesse des systèmes informatiques ainsi spécifiés seront bien supérieures aux critères aujourd'hui courants.

Remerciements

Nous remercions C. Baudoin pour son aide dans la préparation de cet article.

BIBLIOGRAPHIE

[1] ABRIAL Jean-Raymond. – *Z : A Specification Language,* IFIP, Kyoto, Août 1978.

[2] ABRIAL Jean-Raymond. – *Les transformations du langage Z* ; Rapport 1977.

[3] ALFORD MACK W. – A Requirements Engineering Methodology for Real-time Processing Environments, *IEEE Transactions on Software Engineering,* vol. SE-3, pages 60-69, Janvier 1977.

[4] BACKUS John. – Can Programming be liberated from the von Neumann Style ? A Functional Style and its Algebra of Programs *; Communications of the ACM,* vol. 21, n° 8, pages 613-357, Août 1978.

[5] DEMUYNCK Michel et MEYER Bertrand. – *Les langages de spécification : vers une meilleure analyse des problèmes informatiques et des programmes plus fiables ;* Actes de la Convention Informatique, Paris (Palais des Congrès), Septembre 1978.

[6] GUTTAG John. – *The Specification and Application to Programming of Abstract Data Types;* rapport CSRG-59, Université de Toronto, Septembre 1975.

[7] JACKSON M.O. – *Principles of Program Design;* Academic Press, London, 1975.

[8] JENSEN Kathleen et WIRTH Niklaus. – *PASCAL User Manual and Report;* Springer-Verlag (Berlin), 2e édition, 1975.

[9] KIRK H.W. – Use of Decision Tables in Computer Programming; *Communications of the ACM,* vol. 8, n° 1, pages 41-43, Janvier 1965.

[10] LISKOV Barbara H. et ZILLES, Stephen N. – Specifications Techniques for Data Abstractions; *IEEE Transactions of Software Engineering,* vol. SE-1, n° 1, pages 7-18, Mars 1975.

[11] MEYER Bertrand. – Description des Structures de Données ; *Bulletin de la Direction des Etudes et Recherches EDF, série Mathématiques Informatiques,* n° 2, pages 81-90, décembre 1976. Egalement dans : *Bulletin du Groupe Programmation et Langages de l'AFCET* (GROPLAN), n° 2, janvier 1978.

[12] MEYER Bertrand et BAUDOIN Claude. – *Méthodes de programmation;* Eyrolles, Paris, 1978.

[13] PARNAS David L. – A Technique for Software Module Specification with Examples; *Communications of the ACM,* vol. 15, n° 12, pages 1053-1058, Décembre 1972.

[14] PETRI H. – *Kommunikation mit Automaten;* Thèse, Frankfurt-oder-Main, 1962.

[15] ROSS, Douglas T. et al. – Special Section on Requirements Analysis; *IEEE Transactions on Software Engineering,* vol. SE-3, n° 1, pages 2-34, janvier 1977.

[16] STAY J.F. – HIPO and Integrated Program Design; *IBM Systems Journal,* vol. 15, n° 2, pages 143-154, 1976.

[17] TEICHROEW D. et SAYANI H. – *Automation of System Building;* Datamation, pages 25-30, Août 1971.

[18] WARNIER J.D. – Ensemble de manuels : *Entraînement à la Construction des programmes d'Informatique; Entraînement à la Programmatin; L'Organisation des Données d'un Système; Guide LCS; Les Procédures de Traitement et leurs Données,* etc. Editions d'Organisation, Paris.